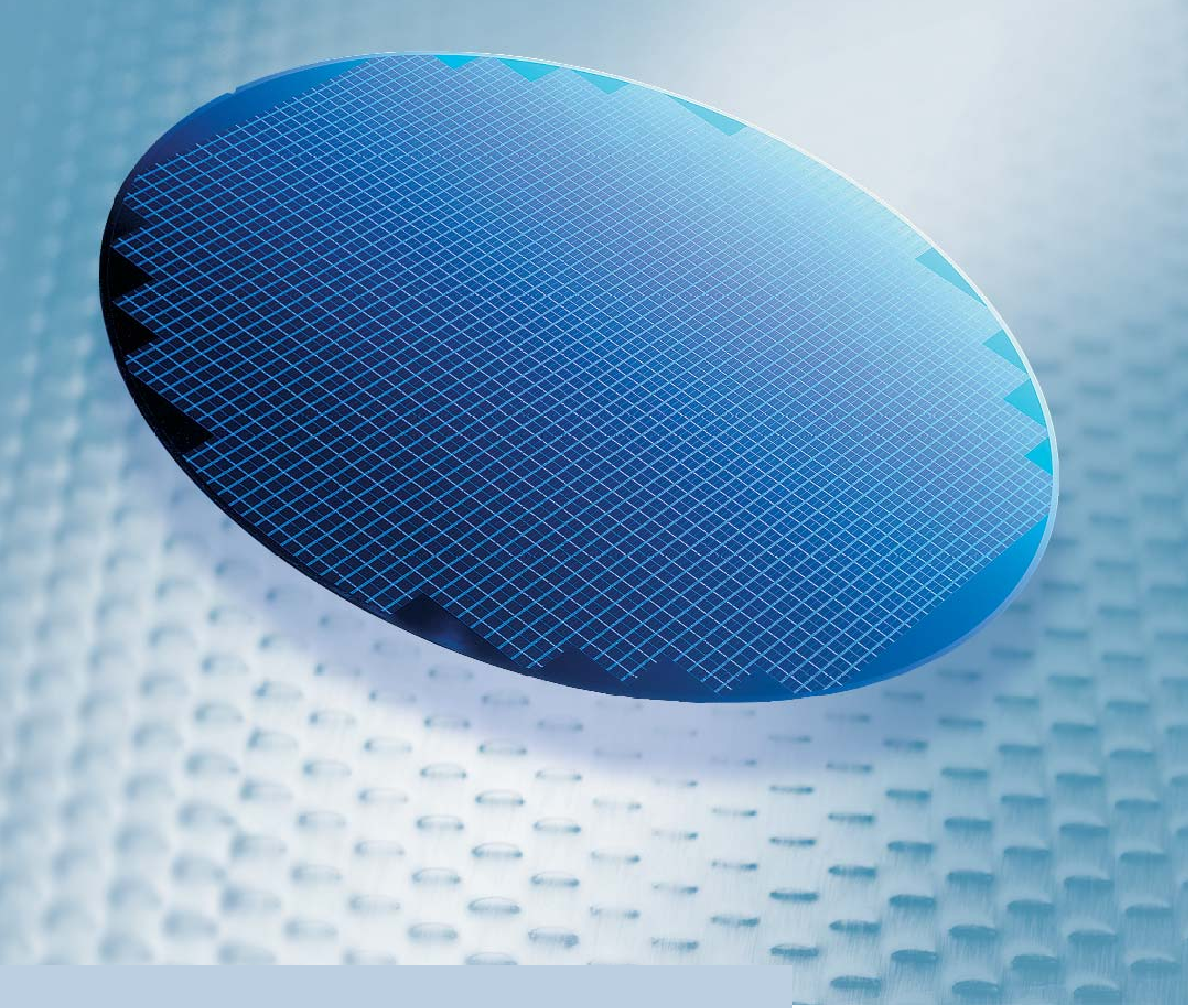
# The INSIDER GUIDE to Planning XC166 Family Designs

## An Engineers Introduction to the XC166 Family



Microcontrollers

Never stop thinking

Edition 2006-02-22

Published by
Infineon Technologies AG
81726 München, Germany



**Attention please!**

The information herein is given to describe certain components and shall not be considered as a guarantee of characteristics.

Terms of delivery and rights to technical change reserved.

We hereby disclaim any and all warranties, including but not limited to warranties of non-infringement, regarding circuits, descriptions and charts stated herein.

**Information**

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office (www.infineon.com).

**Warnings**

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

| **Revision History:** | 2006-02 | V1.0 |
|---|---|---|
| Previous Version: | none | |

| Page | Subjects (major changes since last revision) |
|---|---|
| | |
| | |
| | |
| | |
| | |
| | |
| | |

**We Listen to Your Comments**

Any information within this document that you feel is wrong, unclear or missing at all?
Your feedback will help us to continuously improve the quality of this document.
Please send your proposal (including a reference to this document) to:
**mcdocu.comments@infineon.com**

## Credits

Authors: Michael Beach
David Greenhill
Editor: Alison Wenlock

## Acknowledgements

The authors would like to thank Karl Smith, Mike Copeland and Manfred Choutka of Infineon Technologies plus Joachim Klein of Hitex Development Tools GmbH. for their contributions to this book.

## Preface

This guide contains basic information that is useful when doing your first XC166 family design. There are many simple facts which, if they are known at the outset, can save a lot of time and money. Overall, it is intended to complement the user manuals by putting things into a practical context.

Some of the material can be found in the XC166 family databooks but most of it is simply the result of our practical experience and so is only to be found here. The topics covered are those that are not obvious or are often missed out. Where the user manuals provide a satisfactory explanation, you will be referred back to them, rather than duplicating information here. This is by no means a complete reference work and a lot of additional information can be found on the Infineon website.

**Note:** While every effort has been made to ensure the accuracy of the information contained within this guide, Hitex cannot be held responsible for the consequences of any errors contained therein. Any subjective or anecdotal information presented is not necessarily the official view of either Hitex Development Tools Ltd. or Infineon Technologies AG.

***Prepared By:***
Michael Beach
David Greenhill

***With additional material from:***
Karl Smith, Infineon Technologies UK
Joachim Klein, Hitex Development Tools

## Contents

# 1 RISC Architectures For Embedded Applications

## 1.1 Introduction

*RISC microcontrollers are becoming increasingly popular. The C166S V2 CPU core used in the XC166 series makes extensive use of Reduced Instruction Set Computer (RISC) concepts to achieve its blend of very high performance at modest cost. To understand why RISC techniques are especially suited to high-speed real time embedded systems, it is useful to examine in detail how they grew out of the traditional Complex Instruction Set Computers (CISC) that reached their peak in the late 1980's to mid 1990's. However despite the many inherent advantages of RISC for microcontroller applications, not all RISC microcontrollers fully exploit them for legacy reasons. This section examines some of the critical issues.*

## 1.2 Behind The C166S V2's Near-RISC Core

The quest for ever-greater throughput has been the reason behind the abandonment of traditional Complex Instruction Set Computers (CISC). The demands of workstations involved in CAD tasks and latterly, advanced video games, have been the real driving force behind this. Traditionally, microprocessors have been designed with assembler instruction sets that have been geared towards making the assembler programmer's life easier through the extensive use of microcode to produce ever more powerful instructions. By providing single assembler instructions that perform, for instance, three-operand multiplication, the assembler programmer (and HLL compiler writer) has been relieved of the job of achieving the same result with simpler instructions.

As the CPU needs to be able to recognize and act on (decode) many hundreds of different instructions, it requires complex silicon and many clock cycles. The greater the silicon area, the greater the cost of the device and power consumed. With physical limitations acting to restrict achievable clock speeds on silicon devices, the number of cycles per instruction is obviously very significant in gaining higher performance.

RISCs tend to shift the burden of programming from the microcoder to the assembler programmers and compiler writers. Work within academia and commercial manufacturers has proved that a suitably-programmed RISC machine can achieve a far higher throughput than a CISC for a given clock speed.

Strangely, the mid-range embedded world has been slow to question the suitability of the CISC-based microcontroller. At the very top end, RSIC devices such as ARM9, MIPS and Hyperstone provide stiff competition to the conventional CISC PowerPC and Pentium but for more commonplace embedded tasks, RISC is still relatively uncommon. With the increasing complexity of modern control algorithms, the need for greater processing power is set to become an issue in anything but the simplest applications. In addition, here more than in the workstation world, the worst-case response time to non-deterministic events is crucial, an area where CISCs are especially poor and where most ARM-based RISCs are by no means outstanding.

Many current mid-range 16-bit microcontrollers are based on existing CISC architectures such as the S12, H8, M16C etc., which in common with 8-bit devices such as the 8051, have an internal structure that dates back 20 years or more. With the silicon vendor's need to give existing users an upgrade path, apparently new CPU designs are often based closely on the existing architecture/instruction set, so protecting the user's investment in expensive assembler code.

Like workstations, microcontrollers are programmed in a high level language (HLL) to reduce coding times and enhance maintainability. Inevitably, even with the best compilers, some loss of performance is encountered, emphasizing again the need for improved CPU performance.

In addition to straightforward data processing, microcontrollers must also handle real-world peripherals such as A/D converters, PWMs, timers, Ports, PLLs etc., all of which require real time processing and fast interrupt response.

### 1.2.1 Conventional CISC Bottle-necks

**1. Long And Unpredictable Interrupt Latencies**

Complicated “labour-saving” instructions must hold the CPU’s entire attention during execution, thus preventing real-world generated interrupts from being serviced. Unpredictable latency times result, which can cause serious problems in hard real-time systems. One approach to overcoming the CISC’s poor real-time response has been to bolt a secondary “time processor unit” or other such auxiliary processor onto the core to try and off-load the time-critical portions. However, this results in an awkward design and the need to use a very terse microcode to program it, in addition to the more usual C and assembler for the CISC core itself.

**2. Vast Instruction Sets Give Slow Decoding**

Loaded instructions must be recognised from potentially many hundreds or even thousands of possibilities. Decoding is thus complicated and lengthy.

**3. Frequent Accesses To Slow Memory Devices**

Data is typically fetched from off-chip memory and placed in accumulator-type registers. Mathematical or logical operations are performed and the then is result written back to memory. The value is likely to be required again in the course of the procedure, thus requiring further movements to and from off-chip memory.

**4. Slow Procedure Calling**

When calling subroutines with parameters (essential in good HLL programming), parameters must be individually pushed on to stack. They must then be moved through accumulator register(s) for processing before being returned via stack to caller.

**5. Strictly One Job At A time**

Each peripheral device or interrupt source must have a dedicated service routine which at the very least will require the PSW and PC to be stacked and restored and data removed from or fed to the peripheral device.

**6. Software Has To Be Structured To Suit Architecture.**

Embedded systems frequently contain many separate real time tasks which together form a complete system. Conventional CPUs make switching between tasks slow. Often, many registers have to be stacked to free them up for the incoming task. This problem is aggravated by the use of HLL compilers which tend to use a large number of local variables in library functions which must be preserved.

**7. Redundant Instructions And Addressing Modes**

With the almost universal use of high level languages, compilers are tending to dictate which instructions should be provided in silicon.

In practice, compilers tend to only make use of a small number of addressing modes. This results in a large number of unused addressing modes that serve only to complicate the opcode decoding process.

**8. Inconsistent Instruction Sets**

Instruction sets that have evolved tend to be difficult to use due to large number of different basic types and the inconsistent addressing modes allowed.

**9. Bus Not Fully Utilised**

Whilst complex instructions are being executed, the bus is idle.

## 1.3 The RISC Architecture For Embedded Control

To show how RISC design is used to improve microcontroller throughput, the C166S V2 is used as an example.

Basic Definitions:

1 state time = 1/oscillator frequency

- fundamental unit of time recognised within processor system.

1 machine cycle = state time

- minimum time required to perform the simplest meaningful task within CPU.

The unit of state times is used when making comparisons between RISCs and CISCs as this removes any dependency on clock frequency.

- All state time counts are given in single chip operation mode for both S12X and C166S V2.

### 1.3.1 Bus Interface

To maximise the rate at which instructions are executed, RISC CPUs are very heavily pipelined. Here, on any given machine cycle, up to 4 instructions may be processed by overlapping the various steps. Simplified for clarity, the stages are:

| | |
|---|---|
| **FETCH:** | - get opcode from program store |
| **DECODE:** | - identify opcode from a small list and fetch operands |
| **EXECUTE:** | - perform operation denoted by opcode |
| **WRITE-BACK:** | - result returned to specified location |

Thus although the instruction takes four machine cycles, it is apparently executed in just one (1 state time). Pipelining has considerable benefits for speeding sequential code execution as the bus is guaranteed to be fully occupied.

Some more advanced RISC devices (like the C166S V2) add an extra ADDRESS stage to make a total of 5 pipeline stages and also add a 2-part "PREFETCH" unit:

**Instruction Fetch Unit (IFU)**

| | |
|---|---|
| **PREFETCH:** | - Get instructions from the program memory in the order predicted. Any branches are detected and prediction logic decides if the branches will be taken or not. |
| **FETCH:** | - The address of the next instruction to be fetched is calculated using the branch prediction rules. |

**Pipeline Unit**

| | |
|---|---|
| **DECODE:** | - The instructions are decoded and, if required, the register file is accessed to read the GPR used in indirect addressing modes. |
| **ADDRESS:** | - All the operand addresses are calculated. |
| **MEMORY:** | - All the required operands are fetched from RAM and registers. |
| **EXECUTE:** | - perform the operation denoted by opcode |
| **WRITE BACK:** | - result returned to specified location |

This theoretically gives a doubling in performance on straight-line code. However even though there is a branch prediction unit to minimise the loss of performance caused by branches in real programs, not quite twice the throughput can be achieved.

### 1.3.2 RISC Interrupt Response

In the C166S V2 core, branches to interrupts make use of the injected instruction technique and so vectoring to a service routine is achieved in only 5 machine cycles. The effect of complex but necessary instructions such as MUL and DIV (1 and 21 cycles respectively) stretch this but it is interesting to note that the C166S V2 does provide the DIV as a partially-interruptible instruction. The first 4 cycles lock out all interrupts but the remaining 17 cycles may be interrupted.

Very fast interrupt service is crucial in high-end applications such as engine management systems, servo drives and radar systems where real-world timings are used in DSP-style calculations. As these normally form part of a larger closed control loop, erratic latency times manifest themselves as an undesirable jitter in the controller output.

### 1.3.3 Registers And Multi-Tasking

Traditional microcontrollers have one or more special registers that can be used for mathematical, logical or Boolean operations. In the 8051, there is a single “accumulator” with 8 other registers which may be used for handling local variables or intermediate results in complex calculations. These additional registers are also used to access memory locations via indirect and/or indexed addressing.

As pointed out in items 3 and 4 above, conventional CPUs spend much time moving data from slow memory areas into active registers. The RISC CPU offers a very large number of general purpose registers which may be used for locals, parameters and intermediates. The C166S V2 provides 16 word-wide general purpose registers (GPRs), each of which is effectively an accumulator, indirect pointer and index. With such a large number of GPRs available, it becomes realistic to keep all locals and intermediates within the CPU throughout quite large procedures. This can yield a great increase in speed.

Further significant benefits are derived from the RISC technique of register windowing. As has been said, up to 16 registers are available for use by the program. However, by making the active register bank movable within a larger on-chip RAM, the job of real-time multi-tasking is considerably eased.

Central to this is the concept of a “Context Pointer” (CP), which defines the current absolute base address of the active registerbank in the program memory space. Thus a reference to “R0” means the register at the address indicated by the CP (typically address 0xFD00). Thereafter, the 16 registers originating from CP are accessed by a fast 4-bit offset.

The best example of how the CP is exploited is perhaps a background task and a real-time interrupt co-existing. When the interrupt occurs, rather than pushing all GPRs onto the stack, the CP of the current register bank is stacked and simply switched to a new value, determined at link time, to yield a fresh register bank. This results in a complete context switch in just one instruction but it does rule out the use of recursion.

A hybrid method, which permits re-entrancy, uses the stack pointer to calculate the new CP dynamically. Here, on entering the interrupt, the number of registers now required is subtracted from the current SP and the result placed in CP, with the old CP stacked. Thus the new register bank is located at the top of the old stack, with the old CP and then the new stack following on immediately afterwards. On exiting the interrupt routine, the original registerbank is restored by POPping the old CP from the stack. The SP is reinstated by adding the size of the new register bank onto the current SP.

A further RISC refinement is register window overlapping whereby when a new procedure is called, part of the new register bank defined by CP is coincident with the original at CP:

```
                R3'     ; Register for subroutine's locals and intermediates
                R2'     ; Register for subroutine's locals and intermediates
        R7      R1'     ; Common register, R7 == R1'
CP'     R6      R0'     ; Common register, R6 == R0'
        R5              ; Register for caller's locals and intermediates
        R4              ; Register for caller's locals and intermediates
        R3              ; Register for caller's locals and intermediates
        R2              ; Register for caller's locals and intermediates
        R1              ; Register for caller's locals and intermediates
CP      R0              ; Register for caller's locals and intermediates
```

**MODULE 1**

```
; *** Assignment Of GPRs To Local Variables - Caller ***

x_var   LIT     'R0'            ; Local variable
y_var   LIT     'R1'            ; Local variable

parm1   LIT     'R6'            ; Passed parameter 1
parm2   LIT     'R7'            ; Passed parameter 2

result  LIT     'R6'            ; Value returned from sub routine
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

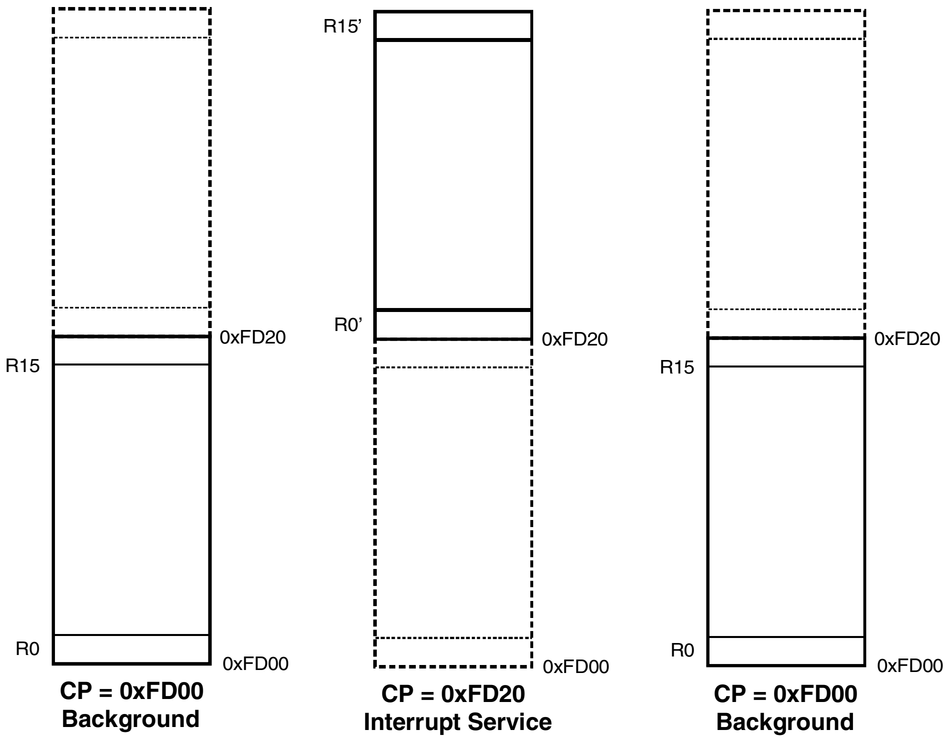
**MODULE 2**

```
; *** Assignment Of GPRs To Local Variables - Sub Routine ***

a_var   LIT     'R2'            ; Local variable
b_var   LIT     'R3'            ; Local variable

input1  LIT     'R0'            ; Received parameter 1
input2  LIT     'R1'            ; Received parameter 2
ret1    LIT     'R0'            ; Final result returned in R0
```

**Fig. A - Giving GPRs Meaningful Names**

The programmer should plan for any value to be passed to the subroutine to be located in the common area, so that all the normal loading and unloading of parameters is avoided. This technique can be used in either absolute or SP-relative registerbank modes.

The ability to switch to a new set of working registers by changing the context pointer register is useful when responding to interrupts, as it avoids the need to save the current registerbank contents before executing the service routine instructions. This can be time-consuming if all 16 registers are to be saved. Typically this requires one instruction (here a switch context "SCXT ") at the start of the interrupt routine to change the registerbank base and a second instruction at the end to restore the original one.




Another method possible on some RISC devices relies on "alternative" register sets that become visible when a predefined interrupt source is activated, so that no saving of current register contents is required. ARM7-TDMI RISC devices typically replace the upper 8 registers in the registerbank when the FIQ (Fast Interrupt reQuest) fires, whereas the C166S V2 has two complete "local" banks of sixteen registers that can be made to appear when either of two interrupt sources is triggered. Unlike conventional registerbanks, these registers do not usually exist in the normal memory space of the CPU and so no context switching is required, thus saving time[1].

To get the best from a RISC's registers, the location of data needs close consideration: although highly orthogonal, the limited number of addressing modes provided for MUL and DIV for example, can appear somewhat restrictive. Fortunately though, most operands involved will already be in registers, so eliminating the need for many addressing techniques. As might be expected, the instructions with the widest range of addressing modes are the simple data moves - the fact that RISCs are the result of very careful analysis of the requirements for fast execution becomes obvious after a short acquaintance!

[1] In fact the current local registerbank is can be given a physical address and made visible by setting the appropriate bits in the BANK field of the PSW register.

## 1.3.4 Coping With RISC Instruction Set (Apparent) Omissions

With largely single machine cycle execution, some conventional "fast" instructions such as CLEAR, INC and DEC become redundant. Therefore, to keep the total number of instructions to a minimum, RISCs simply omit them. Examples are given below:

| Instruction | 80C196 | States | C166S V2 | States |
|---|---|---|---|---|
| Clear Word | CLR | 4 | AND Rn,#0 | 2 |
| Decrement Word | DEC | 4 | SUB Rn,#01 | 2 |
| Increment Word | INC | 4 | ADD Rn,#01 | 2 |

*- all direct addressing mode*

Three-operand instructions are also commonplace in CISCs but not present in RISCs. Although additional instructions are required, the overall number of states is still less than the three-operand CISC equivalent, plus the shorter RISC instructions allow greater opportunity for interrupt servicing.

The following example illustrates this:

Perform: z = x + y

### 80C196 (CISC)

z, x and y are directly addressed memory locations

```
x       DW      1
y       DW      1
z       DW      1

        ADD     z,x,y   ; 5 states - no interrupt possible
```

### C166S V2 (RISC)

z, x and y are memory locations, Rw is a GPR

```
x       DW      1
y       DW      1
z       DW      1

        MOV     Rw,x            ; 2 states
                                ; * Interruptable here
        ADD     Rw,y            ; 2 states
                                ; * Interruptable here
        MOV     z,Rw            ; 2 states
                                ; ____
                                ; 6 states
```

One extra state required when using RISC approach.

However, if the variables are assigned recognising that this is a RISC:

x and y are memory locations, z is a GPR

```
x       DW      1
y       DW      1

z       LIT     'R0'          ; z is assigned to GPR R0 via a LITeral definition

        MOV     z,x           ; 2 states
                              ; * Interruptable here
        ADD     z,y           ; 2 states
                              ; ----
                              ; 4 states
```

- 1 state saved over CISC.  The above was chosen as a worst case RISC, best case CISC example.

For a normal 2 operand ADD, the RISC uses two states compared to the CISC's 4, a 50% improvement.

- Assigning all variables to GPRs would probably make sense in the context of a real program.
- This trivial example shows how familiarity with RISC's programming techniques improves performance.

## 1.4 RISC And Real World Peripherals

Within the workstation or desktop computer RISC, superscalar operation allows parallel execution of instructions, made possible by having discrete addition, multiplication, shift and other dedicated units, each with their own pipelines.

No RISC microcontroller (yet) quite offers this (although the 32-bit Tricore is heading this way) but something similar is possible to service on-chip peripherals such as an A/D converter.

A common situation occurs in conventional microcontrollers whereby some regular event requires attention from the CPU to load or unload data.  Typically, an A/D converter will cyclically read a number of channels, causing an interrupt when completed or simply waiting for the CPU to poll its status.  The net result is the valuable CPU time is spent doing what even for a microcontroller is a simple, repetitive task.

The RISC C166S V2 allows the interrupt service routine to be serviced and completed in a single machine cycle (via the "Peripheral Event Controller", described in section 8.4).  In the case of a periodic A/D conversion, on each conversion the result from the "ADDAT" register is stored in a table from where the readings may be later retrieved by the CPU.  This mechanism requires the CPU to perform only a one single-cycle instruction equivalent to "MOV  [table_addr+],ADDAT" after each conversion.  At the end of the table, a traditional interrupt routine is required to reset the table pointer to permit another series of conversions and automatic result transfers.

Any real-world generated data can be handled in this way, leaving the CPU free for data processing rather than simple data collection.

For many applications, a close coupling between the CPU and IO pins is useful for the detection and generation of real time pulses.  Traditional RISC CPUs often have lengthy and unpredictable delays between for example, software setting a port pin and the actual pin changing state.  For input, the reverse situation applies.  Such delays are caused by the RISC core not being adapted to hard real-time use and communicating with peripherals through the VLSI (very large scale integration) bus.

The C166S V2, being designed as a real-time controller does not suffer from this kind of problem and there is a direct connection between the CPU core and the peripheral set.

### 1.4.1 RISC Benefits In Embedded Applications

**1. Near-DSP throughput**

For example, the XC166 can achieve 20 million instructions per second (20MIPS) at a 20MHz clock (50ns machine cycle time). At 40MHz this rises to 40MIPS with a 25ns cycle time. This is a result of pipelining and the ability to contain the active data for entire procedures within the CPU registers.

**2. Simpler Assembler Coding**

Although the instruction set is less diverse, the consistency of addressing modes makes assembler coding easier.

**3. Very Fast Response To Non-Deterministic Events**

By eliminating instructions that take many cycles, interrupt response is improved. Smaller instructions effectively yield higher "sampling rate" for real world events.

**4. Single Machine Cycle Context Switching**

By careful use of multiple register banks controlled by a base pointer, context switching in a multitasking system can be performed in just one instruction.

In addition, parameter passing overhead to subroutines is eliminated by use of overlapping register windows, so that parameters lie in the common area.

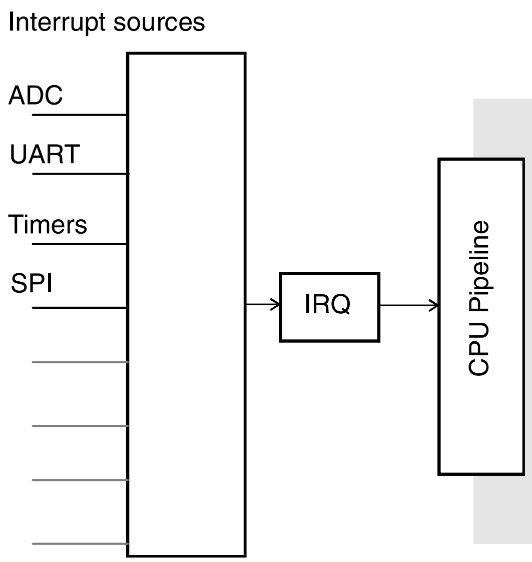
**5. Alternate or Local Registerbanks**

By planning the use of interrupts carefully, zero-time context switching is possible.

## 1.5 Traditional RISC v New RISC

Although RISC is inherently conducive to fast interrupt response/low latency, the legacy of having RISC designs with their roots in 1980s desktop computers means that although the handling of single interrupts may be fast and reasonably deterministic, with interrupt-intensive applications, some problems can be experienced.

Interrupt systems have changed a lot since the days of a single IRQ input with multiple sources requesting interrupt service through it. Here the single interrupt vector required the user to check the source of the interrupt before servicing it. This ruled out the nesting of interrupts with the result that in systems with significant interrupt activity, effective latency times could become very long and completely unpredictable, despite the very high straight line speed of the CPU core.

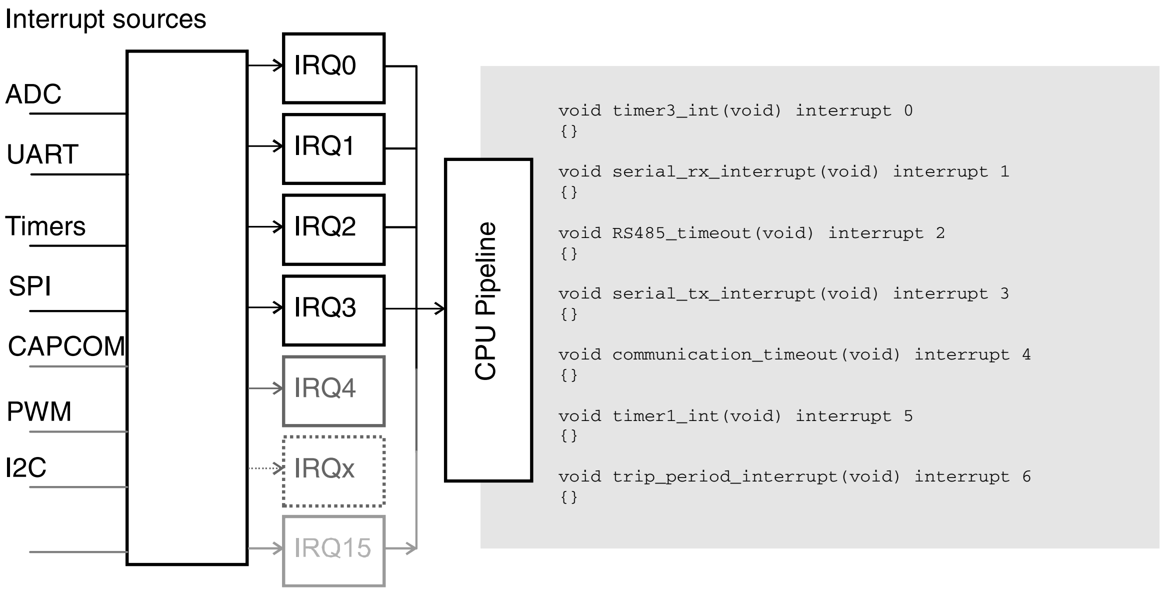



```
//
// Non-Vectored Interrupt Service Routine
//

void NonVectoredIRQ (void) __attribute__ ((interrupt("IRQ")))
{
    // Test for the interrupt source

    If(VICIRQStatus & 0x00008000)
    {
         // Set the LED pins

       IOSET1    = 0x00FF0000;

         // Clear the peripheral interrupt flag

       EXTINT  = 0x00000002;
    }

    // Dummy write to signal end of interrupt
    VICVectAddr = 0x00000000;
```

**Interrupt Request Handling in 1980's Style RISC CPU**

However, many traditional RISC CPUs still have interrupt management of this type (or some variation of it) which unfortunately negates the inherent advantages of fast interrupt response conferred by the RISC approach. The net effect is that although the best-case interrupt latency is good, the worst case figure is almost impossible to predict. Thus it cannot be taken for granted that all RISC microcontroller CPUs will be suitable for hard real-time systems.

Later RISC devices like the C166S V2 use a more up to date approach to interrupt handling, designed to make best use of the RISC inherently short interrupt latencies.  Here interrupt sources can be freely assigned to up to 15 different levels, which are prioritised to allow full interrupt nesting.  This allows the short interrupt latency times permitted by RISC to be fully exploited and both the best and worst case latencies can be estimated with a high degree of certainty.




**Interrupt Request Handling In The XC16x RISC CPU**

# 2 Getting Started With The XC166

## 2.1 Basic Considerations

### 2.1.1 Family Overview

The XC166 family currently includes three main variants, the XC161, the XC164 and the XC167, although with different memory and peripheral options this adds up to considerably more in reality. According to the XC166 User Manual, the XC family is the 4$^{th}$ generation of Infineon's 16-bit Microcontroller. The first family member was the 80C166, available in masked ROM, FLASH EPROM (88C166) and ROMless versions. The second member was the C167 which had an expanded addressing capability, integral chip selects plus many more peripherals and introduced some new assembler instructions. The third generation added flexible power management. Although the XC is binary compatible with the C167 ('Classic 167'), many enhancements have been made to increase performance. The XC166 family uses the C166S V2 core which includes a number of important enhancements. The MAC-unit adds DSP-functionality to handle digital filter algorithms and greatly reduces the execution time of multiplications. The 5-stage pipeline, single-cycle execution of most instructions, and PEC-transfers within the complete addressing range increase system performance. Debugging the target system is supported by integrated functions for On-Chip Debug Support (OCDS). The other major enhancement is the addition of full automotive spec on-chip FLASH.

### 2.1.2 Fundamental Design Factors

When starting out on a XC166 family design, there are a number of basic things you must decide. Wrong decisions here can have expensive consequences later in the project. There are a good many features of the architecture that can be a bit puzzling to those used to conventional devices. What follows is a simple guide to what you really need to know to get the best from this ingenious and powerful microcontroller family!

- *What clock speed is required to achieve the necessary CPU processing power?*
- *What sort of clock source is suitable?*
- *What sort of reset circuit should be used?*
- *What CPU sockets are available*
- *How is the CPU configured?*
- *Will the CPU boot into internal or external ROM?*
- *How is the on-chip FLASH EPROM to be programmed?*
- *How is external FLASH EPROM to be programmed?*
- *How can any external memory be added?*
- *Is a full 16-bit external bus necessary or will an 8-bit bus be sufficient?*
- *Is an external bus required at all?*
- *Will there be some external peripheral chips that will require different bus modes?*
- *How much IO is required to implement the application?*
- *Should WRH/WRL be used?*
- *Should the chip selects be used?*
- *Which peripheral pins are best allocated to the various different signal processing or generation functions in the application?*
- *And many others...*

### 2.1.3 Setting The CPU Hardware Configuration Options

In common with many modern microcontrollers, between the /RESIN pin going high and the rising edge of the first ALE pulse, the XC reads its start up configuration. This is read from one of two places depending on whether the processor is booting from internal or external memory. This is determined from the level of the /EA pin at start up. If /EA is High, then the processor will boot from internal memory. In this case the startup configuration is read from the RD, WR and ALE pins. With only three pins available, only a minimum configuration can be achieved, the configuration is completed in software during the start up.

The following settings are available for an internal start:

* *Boot from the standard or alternate reset vector*
* *Enable the standard or alternate bootstrap mode*
* *Use P20.12 as either /RSTOUT or GPIO*

If /EA is Low, then the processor will boot with the external bus enabled. The configuration is read from the bit pattern on Port 0 to determine the following fundamental settings:

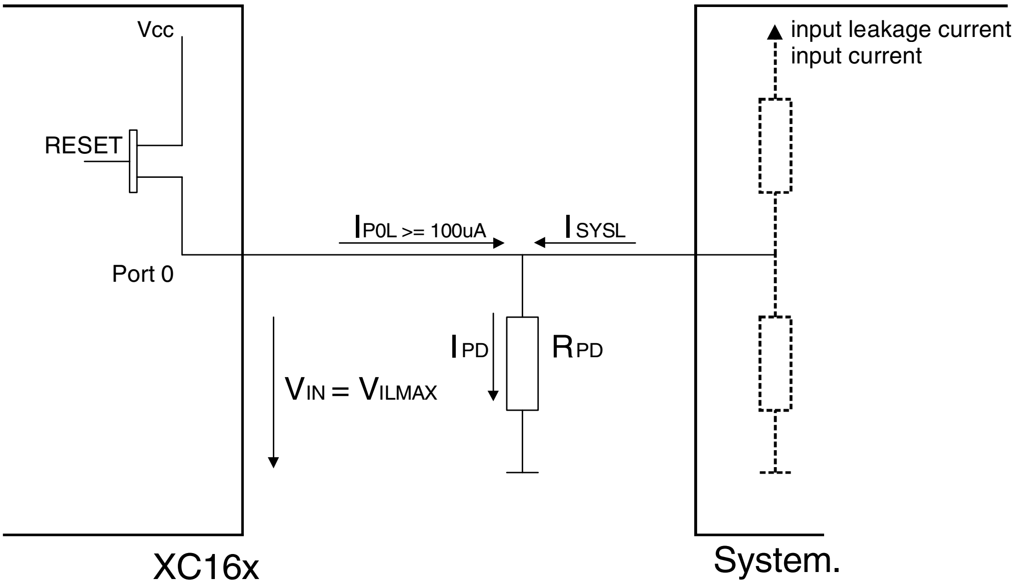
* *What the default bus mode is*
* *How many pins on port 6 should be used as chip selects*
* *How many segment address lines should be used*
* *Whether the on-circuit emulation mode is to be entered*
* *Whether the WRITEHIGH/WRITELOW mode required*
* *Whether the BOOTSTRAP mode is to be activated*
* *What clock factor is to be used*

The pattern is placed onto the port by the user attaching pull-down resistors to the appropriate pins. For example, to get the processor booting from internal memory and in bootstrap mode, /EA will be High and /RD will need to be pulled Low through a resistor. To set 16 bit non-multiplexed bus mode from an external start, /EA will be Low and a pull-down resistor is added to Port 0.7, while Port 0.6 floats high. The values of the pull-down resistors should be calculated with reference to the overall loading on Port 0, from external memory devices etc., using the formulae given in section 2.2. The value required for a typical 1 EPROM + 1 RAM system is 8K0, this representing the stated maximum value. It covers 90% of all designs seen to date. In extreme cases, as little as 1K8 can be used but this is exceptional as the leakage currents from modern memory devices are extremely small. Overall, the user is simply advised to check the situation in the design and not to just to blindly accept the usual 4k7 value!

**Note:** The databooks frequently refer to port 0 either as a 16-bit port or as two 8-bit ports, made up of Port 0L (LOW) and Port 0H (HIGH). Thus Port 0.15 is bit-16 on port 0 which is also Port 0H.7. By the same convention, Port 0.7 is also known as Port 0L.7.

## 2.2 Calculating The Pull-Down Resistor Values

Finding the value of the pull-down resistors for your design is fairly straightforward. You will need to know the leakage current from the devices such as RAMs, ROMs etc that are attached to the bus.




**Pull-Down Resistor Current Flow**

$V_{ILMAX}$ = Highest voltage that will be accepted as a ‘0’
$I_{SYSL}$ = Leakage current from RAMs, ROMs etc.
$I_{P0L}$ = Current flow from XC166’s Port 0 when pin is at $V_{ILMAX}$
$R_{PD}$ = Pull down resistor on Port 0

***From XC166 Datasheet:***

$V_{ILMAX}$ = (0.2 x Vcc) - 0.1V => 0.8V <= $V_{ILMAX}$ =< 1.0V
Vcc = 5V +/-10% => 4.5V =< Vcc =< 5.5V

Pull Down Resistor Calculation

$$R_{PD} < \frac{V_{ILMAX}}{I_{PD}} = \frac{V_{ILMAX}}{I_{P0L} + I_{SYSL}}$$

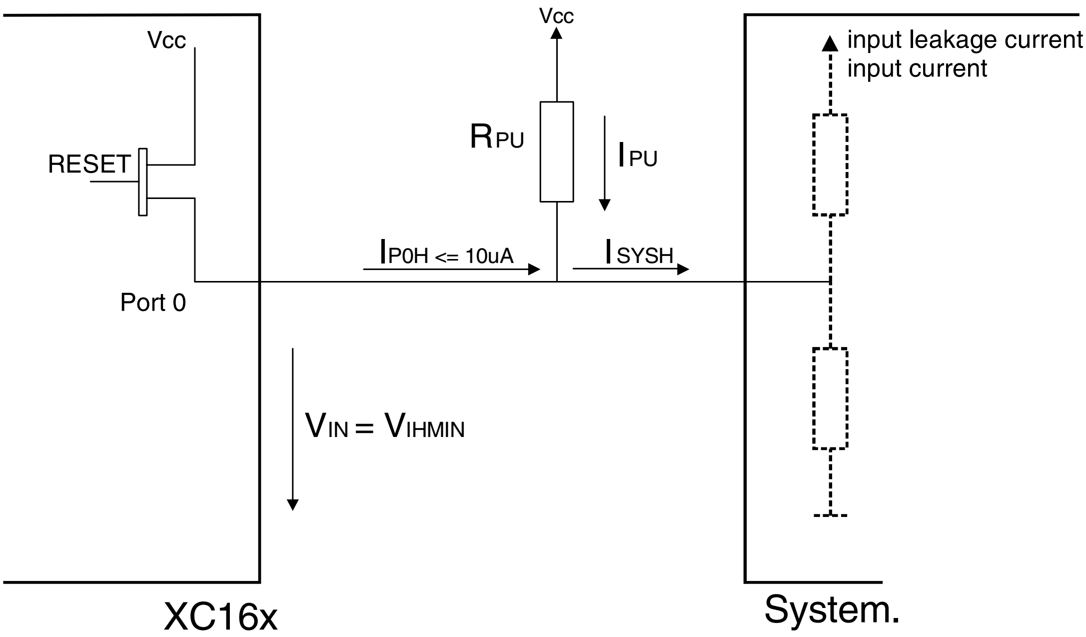
Example Without System Leakage Current, $I_{SYSL}$:

$$R_{PD} < \frac{V_{ILMAX}}{I_{P0L}} = \frac{0.8V}{100uA} = 8K0$$

Thus the maximum recommended value is $R_{PD}$ = 8K0. In practice, 5K6 to 8K2 is almost always used, the former value taking account of typical leakage currents from memory devices on the bus.

### 2.2.1 Pull-Up Resistor Calculations

In some designs, the loading on the bus can be such that there is a net flow of current into the external devices to ground, i.e. the bus sinks current. In extreme cases, this can cause the Port 0 pattern read by the XC166 to be incorrect. It must be stressed that this is very rare but can easily be compensated for by using a high-value pull-up resistor. Such measures are only required if the current sunk into the external device $I_{SYSH}$, is greater or equal to 10uA. Before finalising any design the condition should be checked for and a pull-up resistor added if necessary. The procedure for calculating the pull-up resistor is as follows:




**Pull-Up Resistors On Port 0**

$V_{IHMIN}$ = Lowest voltage on pin that will be accepted as a '1'
$I_{SYSH}$ = Current sunk into bus devices etc.
$I_{P0H}$ = Current that can be drawn from XC166's Port 0 at $V_{IHMIN}$
$R_{PU}$ = Pull up resistor on P0

***From XC166 Datasheet:***

$V_{IHMIN}$ = 0.2 x Vcc + 0.9v - 0,1V => 1.8V <= $V_{IHMIN}$ =< 2.0V
Vcc = 5V +/-10% => 4.5V =< VCC =< 5.5V

Pull Up Resistor Calculation

$$R_{Pu} < \frac{V_{PU}}{I_{PD}} = \frac{V_{CCMIN} - V_{IHMIN}}{I_{SYSH} - I_{P0H}}$$

Example: $I_{SYSL}$ = 50uA

$$R_{PU} < \frac{4.5v - 1.8v}{50uA - 10uA} = 67.6K$$

# 2.3 Start-Up Configuration

## 2.3.1 Internal Start Configuration

If /EA is High, the processor boots from internal memory. This diagram shows the possible configuration functions:

| /EA | /RD | ALE | /WR | Description |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | Standard start, ASC0 bootloader enabled (Addr. C0'0000$_H$), use P20.12 as GPIO |
| 1 | 0 | 0 | 1 | Standard start, ASC0 bootloader enabled (Addr. C0'0000$_H$), use P20.12 as RSTOUT |
| 1 | 0 | 1 | 0 | Alternate start (CAN) bootloader enabled (Addr. C1'0000$_H$), use P20.12 as GPIO |
| 1 | 0 | 1 | 1 | Alternate start (CAN) bootloader enabled (Addr. C1'0000$_H$), use P20.12 as RSTOUT |
| 1 | 1 | 0 | 0 | Standard internal Start (Addr. C0'0000$_H$), use P20.12 as GPIO |
| 1 | 1 | 0 | 1 | Standard internal Start (Addr. C0'0000$_H$), use P20.12 as RSTOUT |
| 1 | 1 | 1 | 0 | Alternate internal Start (Addr. C1'0000$_H$), use P20.12 as GPIO |
| 1 | 1 | 1 | 1 | Alternate internal Start (Addr. C1'0000$_H$), use P20.12 as RSTOUT |

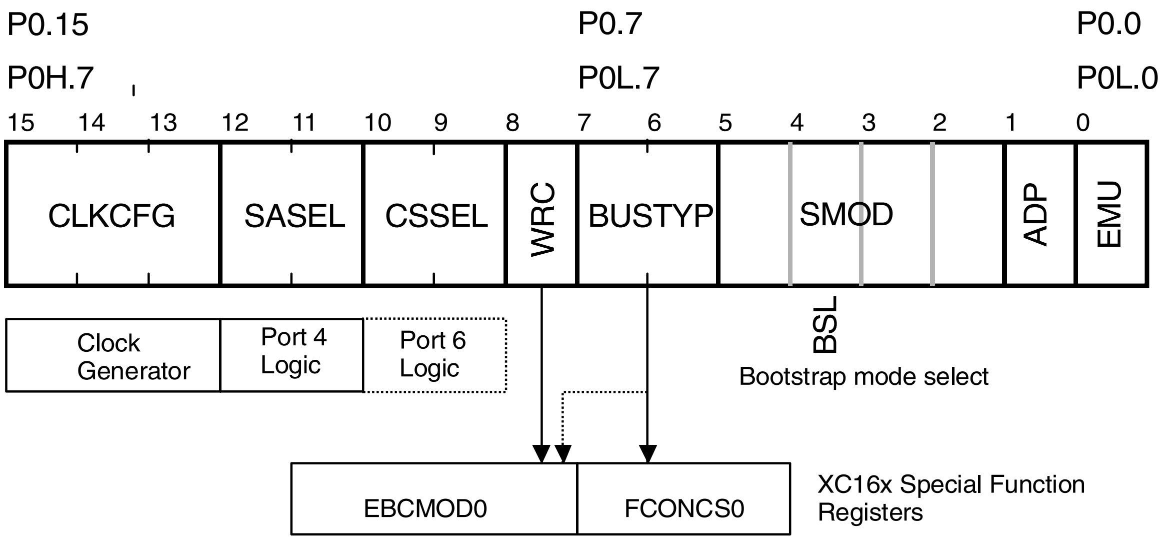
All other startup configuration initially defaults to a 'safe' worst case mode. The clock generation in bypass mode with a 2:1 factor ensuring proper operation for the defined input frequency range of up to 50MHz. The RSTCFG register default vale is 0x0DFF. Changes to this configuration can be made in software.

Another consideration that should be made is with regard to the /EA pin itself. Although this pin needs to be High for internal start, the recommendation is that this should be pulled high through a resistor rather than being connected directly to the rail. This is discussed in more detail in chapter 11, but should it become necessary to connect a full in-circuit emulator to a board, the emulator will need to be able to pull /EA low.

In the majority of XC166 designs, the internal start mode should be used. If an external bus is required then this can be enabled through software running from the internal FLASH.

## 2.3.2 External Start Configuration

This diagram gives the individual configuration functions of the Port 0 pins when the CPU is between the end of reset and the rising edge of the first ALE:




**Port 0 Pin Functions – RSTCFG Register Layout**

**ROC** - When pulled Low, /RSTOUT is deactivated automatically at the end or reset, otherwise it is deactivated by user software

**ADP** - On-circuit emulation mode puts all the XC's pins into a high-impedance tristate condition so that an emulator's clip-over adaptor can be attached to a soldered-in device. Note that if the clock source is a crystal, pin XTAL2 must be disconnected from the processor so that the emulator's CPU can pick up the clock. DO NOT FIT A PULL DOWN RESISTOR ON THIS PIN!

**SMOD** - Special Modes for bootstrap loader

| P0L.5 | P0L.4 | P0L.3 | P0L.2 | Boot Mode |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | Alternate start – 0xC10000 |
| 1 | 0 | 0 | 1 | Alternate TwinCAN bootstrap mode – 0xC10000 |
| 1 | 0 | 1 | 1 | Standard ASC0 bootstrap mode – 0xC00000 |
| 1 | 1 | 1 | 1 | Standard start (default) – 0xC00000 |
| 1 | 0 | 0 | 0 | Alternate SSC0 bootstrap mode – 0xC00000 |

*All other combinations are reserved for future use.*

**BUSTYP** - The external bus type can be set as shown below. These two pins form the BUSTYP field in the FCONCS0 special function register, where it can be modified by software.

| P0L.7 | P0L.6 | External Bus Mode |
|---|---|---|
| 0 | 0 | 8-bit non-multiplexed |
| 0 | 1 | 8-bit multiplexed |
| 1 | 0 | 16-bit non-multiplexed |
| 1 | 1 | 16-bit multiplexed (default) |

**WRC** - Cause the /WR pin to become /WRH (write high) and /BHE to become /WRL (write low) to make the use of 8-bit RAMs in a 16-bit system easier. See section 4.2.

**CSSEL** - The number of chip selects that are to be enabled on Port 6 (or Port 4 on the XC164)

| P0H.2 | P0H.1 | Chip Select Lines |
|---|---|---|
| 0 | 0 | Three: /CS2, /CS1, /CS0 |
| 0 | 1 | Two: /CS1, /CS0 |
| 1 | 0 | None: |
| 1 | 1 | Four: /CS3, /CS2, /CS1, /CS0 (default) |

**SALSEL** - Number of “segment address” lines, i.e. how many additional address lines above A15 will be enabled.

| P0H.4 | P0H.3 | Segment Address Lines On Port 4 |
|---|---|---|
| 0 | 0 | Four: A16 - A19 |
| 0 | 1 | None: |
| 1 | 0 | Eight: A16 - A23 |
| 1 | 1 | Two: A16, A17 (default) |

**CLKCFG** - Programming for processor clock input, with optional phase lock loop (PLL) clock multiplier.

| P0H.7 | P0H.6 | P0H.5 | Clock Generator | Frequency Multiplier Control |
|---|---|---|---|---|
| 0 | 0 | 0 | fmc = fosc /2, | fosc = 1 – 50MHz |
| 0 | 0 | 1 | fmc = fosc x 2.5, | fosc = 12 – 16MHz |
| 0 | 1 | 0 | fmc = fosc x 2.5, | fosc = 8 – 12MHz |
| 0 | 1 | 1 | fmc = fosc, | fosc = 1 – 40MHz |
| 1 | 0 | 0 | fmc = fosc x 5, | fosc = 4 – 6MHz |
| 1 | 0 | 1 | fmc = fosc x 2, | fosc = 12.5 – 18.7MHz |
| 1 | 1 | 0 | fmc = fosc x 4.5, | fosc = 5.6 – 8.3MHz |
| 1 | 1 | 1 | fmc = fosc x 3, | fosc = 8.3 – 12.5MHz (default) |

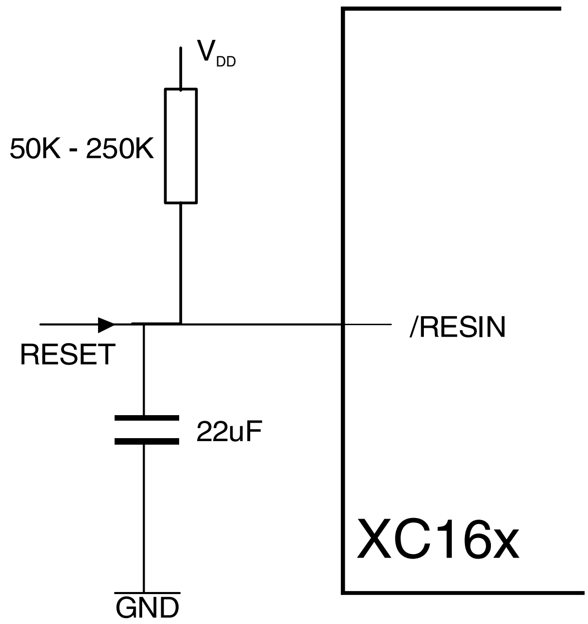
## 2.4 Reset Control

The XC family has two reset pins, /RSTIN and /RSTOUT. The former is a conventional active-low reset input while /RSTOUT is an output, the operation of which is configurable. For single chip applications /RSTOUT will probably not be needed and consequently can be configured as a general purpose IO pin. If it is being used, it will go low at the same time as /RSTIN and stays low either by software when the CPU executes the EINIT (end-of-initialisation) instruction, or until it is deactivated automatically at the end of the internal reset. /RESOUT is thus a means of keeping peripheral devices in a reset state until the CPU is fully initialised.

The /RESIN input must be kept low until the power supply has reached 2.25v for the Vddi rail and 4.5v for Vddp. Once stable, any low level on /RSTIN of more than two state times (50ns @ 40MHz) will reset the CPU. Low times of less than this must be avoided.

The pin has no internal pull-up resistance, so the simplest reset circuit is a pull up resistor and a capacitor to ground. The value must be chosen to give a time constant long enough for the power supply to stabilize.

However, such a simple arrangement is not suitable for use in those situations where the power supply could suffer from instability or brown-outs. In most commercial products, the use of a proper microprocessor power supply and RESET manager such as the TLE7469 is highly recommended. This device provides both the 5v and 2.5v supplies and the CPU's RESET.

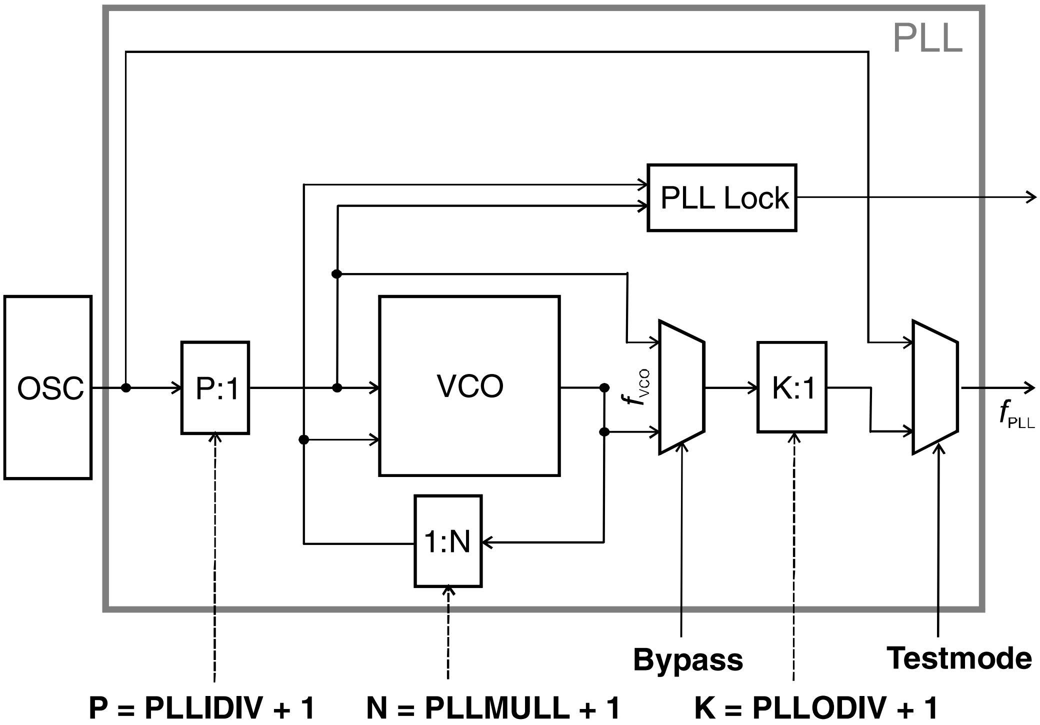



**Very Simple Reset Scheme**

## 2.5 Clock Speeds And Sources

The basic unit of time in the C166S V2 core is a single state time, corresponding to 25ns at 40MHz. Most instructions execute in one state time, i.e. 25ns. Oscillator modules must have a rise and fall time of better than 8ns.

The XC166 is equipped with a PLL system to generate the clock from an external crystal. Exactly how this is handled is determined by the boot mode. The PLL has a series of multipliers and dividers which can be configured by the user directly via the PLL control registers or indirectly via the top three Port0H configuration bits.

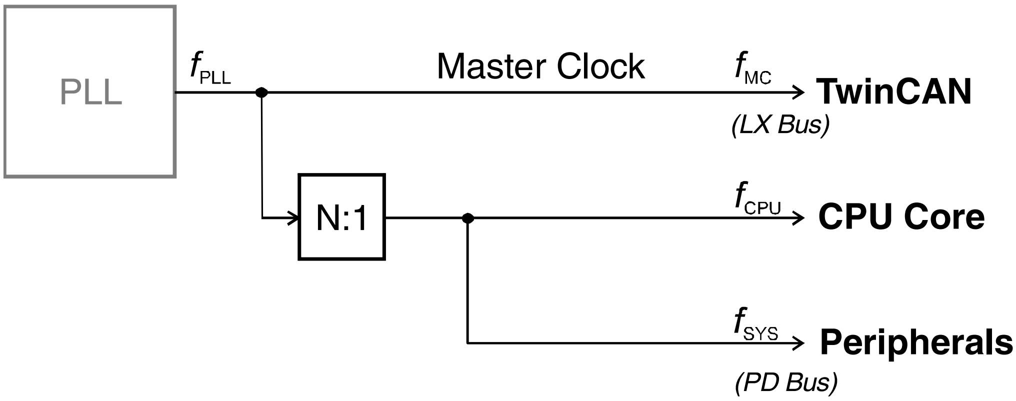



**Internal Clock Distribution Scheme**

The output frequency $f_{PLL}$ is calculated as:

$f_{PLL} = fosc * N/(P * K)$

This signal is then used to drive the whole XC166 device. The only notable exception is the real time clock, which may be driven either from main system clock or the auxiliary oscillator. One Master clock period is known as "$T_{CM}$".




**PLL Output Distribution To XC16x Modules**

The clock is distributed from the PLL as shown. The clock to the CPU and peripherals may be further divided by up to a factor of N = 2 using the CPSYS bit in the SYSCON1 register, although this is rarely necessary.

**Relationship between internal clocks**

$f_{CPU} = f_{PLL}$ / N
$f_{SYS} = f_{PLL}$ / N
$f_{MC} = f_{PLL}$

## 2.5.1 PLL Start Up

The PLL will provide a stable clock even if there is no external crystal or oscillator fitted so that the CPU can run, albeit at a low frequency. In fact this clock is used until the PLL has synchronized to any external oscillator. The base frequency is selected in software by the PLL VCO band select (PLLVB field in PLLCON) which defaults to 20MHz. The K factor (PLL output divider) is set to divide by 16 under these conditions so that the CPU will be running at 3.75MHz.

The PLL starts to run once the 2.5v rail reaches around 1.5v. It is running at 3.75MHz at this point and begins monitoring the incoming oscillator signal for stability over 2000 cycles before attempting to synchronize. If the processor is in bootstrap mode, there is an internal timeout of 30ms after which no further attempt to synchronize is made and the PLL remains at the base frequency. Under these conditions, Baud rates above 9600 are unlikely to work.

PLL bypass mode can give very low clock speeds to reduce power consumption. Here just the K & P dividers are used to give a maximum division of 60.

## 2.5.2 External Bus Start

There are very few cases where the XC166 would be started in external bus mode. In almost all cases, the internal start should be used!

For external start, the state of Port0H pins are read coming out of RESET, as explained in the previous section. The value of the top three bits is used to form a value that is written into the RSTCFG register. This value is the used automatically to configure the clock multiplier. If the pins are not pulled-down then the multiplier is set to divide by 2, as with classic C167 devices. However when planning the pull-down resistor configuration, the user must make sure that the input crystal frequency is within the acceptable range for the required CPU frequency. For example, to use the x5 multiplier, the input frequency must be in the range of 4-6MHz (see the CLKCFG table in section 2.3.2). The reason for these limited ranges can be seen when the internal structure of the PLL is examined in detail – see below.

The user may also select suitable values of P, K & N directly to get the required multiplier by changing the value of the PLLCON register. This procedure is covered in detail in the next section.

### 2.5.3 Internal ROM Start

In this mode, the Port0H pins are not examined for any pattern and it is up to the user to set the clock multiplier manually. The default clock multiplier will be divided by 2 which typically means that on an 8MHz crystal, the CPU will only be running at 4MHz. This can have some side effects, such as preventing the on-chip bootstrap loader from being able to auto-baudrate-detect at greater than 19200 Baud. This is a problem if any FLASH programmer (such as MEMTOOL or FLASHXC.DLL) using the ASC0 bootstrap mode wants to run at 115200 Baud to minimize download time.

Any change to PLLCON aimed at changing the clock multiplier will require the PLL to re-lock. This typically takes around 30us but a maximum of 200us is possible worst-case.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **PLL WRI** | **PLL CTRL** | | **PLLMUL** | | | | | **PLLVB** | | **PLLIDIV** | | **PLLODIV** | | | |

Note that the values of K, N & P put into this register must have one subtracted from them i.e.:

PLLMUL = N – 1<br>PLLODIV = K – 1<br>PLLIDIV = P - 1

**Note:** The input divider “P” must be at least 2 (divide by 2) if the input clock does not have a guaranteed 50/50 duty ratio.

### 2.5.4 Choice Of Clock Speed

As the XC166 is really intended as a single-chip microcontroller with code storage in internal FLASH, there is no real reason to run at anything other than the full 40MHz permitted by current silicon. Access to external RAM and ROM may require waitstates (see chapter 3) but assuming that all time-critical code execution will be from internal FLASH then it makes sense to run as fast as possible.

Generally the XC166 family needs 1 wait state for 40 MHz operation and 0 waitstates for 20 MHz operation. However there is an exception for XC166-32F devices which come into two versions – “Grade A” and “Standard”. Grade A devices support 1 waitstate operation at 40 MHz and 0 waitstates at 20 MHz. Standard devices need an additional wait state (i.e. 2 waitstates) at 40 MHz and 1 waitstate at 20 MHz.

The addition of the waitstate to the internal FLASH causes on average a 5-15% performance reduction in typical applications. The 5-stage pipeline combined with the 64-bit FETCH from the on-chip FLASH mean that up to 2 waitstates can be added without prejudicing throughput on linear code, any loss being due to the number of wrongly-predicted branches that will occur in real programs.

If the system contains external FLASH then the number of waitstates is set independently using the external chip select control registers.

### 2.5.4.1 Choosing The N, K & P Values

For any combination of oscillator frequency and required CPU frequency (Master Clock when CPSYS = 0), there are usually several combinations of K, P & N that will give the required result. The question is therefore how to choose the best combination of values. As the Master Clock is usually driven from the PLL, there is inevitably some jitter on the output as the PLL VCO is continuously adjusted to keep the frequency at the required value. The PLL design is such that abrupt changes in frequency are prevented but there is a small cycle-to-cycle variation in period (the jitter) that needs to be considered in the hardware design.

### 2.5.4.2 Implications of Jitter

Jitter on the master clock can affect the timings of:

- Memory accesses
- CAN bit period
- Peripheral timebase accuracy

However in almost all situations the level of jitter from the XC166 PLL system extremely small but it still ought to be taken into account. The quality of the clock source also should be considered, as it is common for poorly-designed crystal circuits (see later) to cause significant variation in the period-to-period timing. In extreme cases, a poor clock combined with a non-optimal choice of PLLCON parameters can cause transmission errors on the CAN peripheral. In addition, memory access calculations can be upset as the jitter can reduce the bus cycle time and thus increase the apparent clock frequency so that data is misread. Timing errors are less of a problem with peripherals such as the CAPCOM or ASC0 as they tend to deal with time periods in the microsecond to millisecond range. If the oscillator design is good and the K, N & P values properly chosen, then PLL jitter effects on the CAN module can be regarded as insignificant, although they still need to be examined in the light of any external bus devices that may be present in the design.

As an example of how calculated memory access times must allow for PLL jitter, consider the jitter across one bus cycle:

One bus cycle requires = 4 $T_{CM}$ (or $T_{CP}$ if the CPUSYS divider is 1). From the XC166 datasheet:

Jitter (N)(ns) = +/-(1.5 + 6.32 * (N)/$f_{MC}$)

Where N = Number Of $T_{CM}$'s in period over which jitter is to be estimated.

Note that this formula assumes that the highest possible K factor is used (output divider).

For 4 $T_{CM}$'s at 40MHz,

```
Jitter = +/-(1.5 + 6.32 * 4/40)
```

```
Jitter = +/- 2.132ns
```

Any other K value would increase jitter by an indeterminate amount. Therefore it is important to choose the highest possible K. Thus when calculating the bus timings, this 2.132ns must be added to the worst-case PhaseE. In addition, any oscillator tolerance must be allowed for in a similar way. This tolerance will be the sum of the frequency, temperature and the ageing tolerances and is typically not to be more than around 0.02% per period.

### 2.5.5 Choosing The PLLCON Values

Knowing that jitter is significant, how does its reduction influence the choice of PLLCON parameters?

Example:

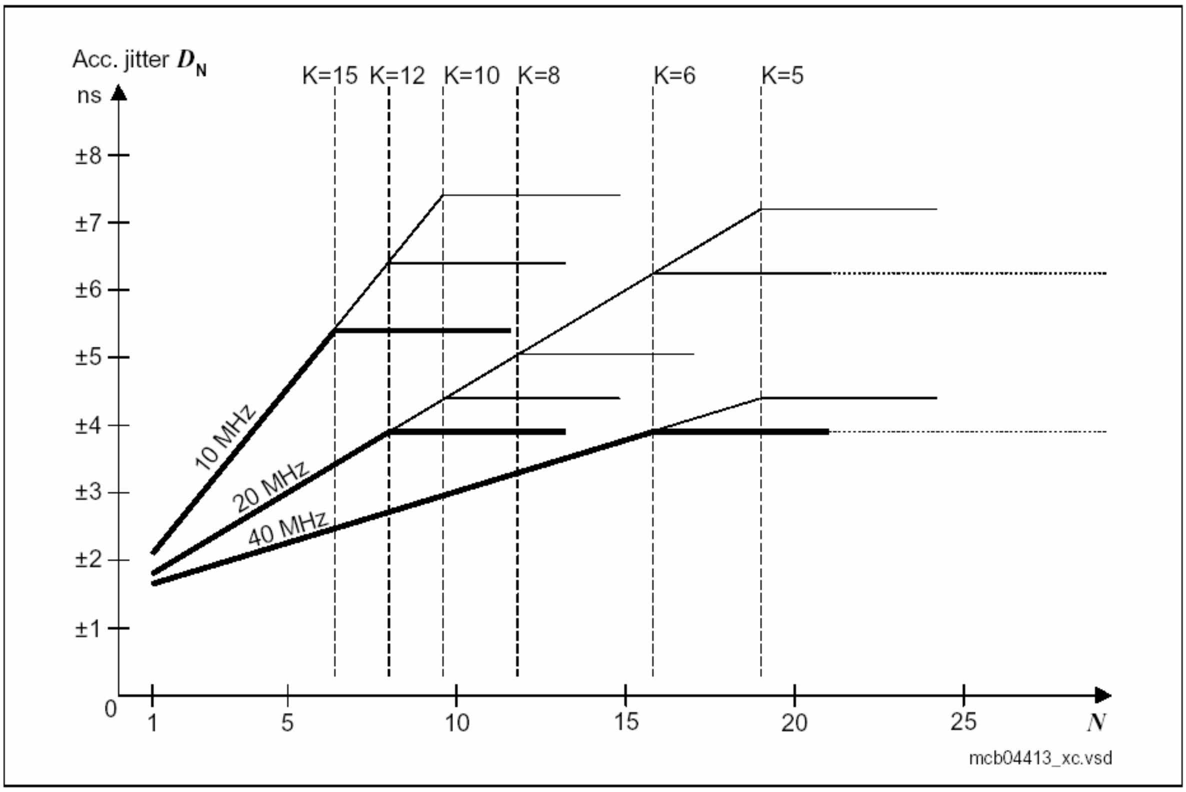
As above, XTAL frequency = 8MHz, Master Clock frequency $f_{MC}$ = 40MHz. This gives at least three possible combinations of N, K & P, as shown.

| (i) PLLCON = 0x7884 *(default value from DaVE)* | (ii) PLLCON = 0x7D12 | (iii) PLLCON = 0x7D85 |
|---|---|---|
| N = 25 | N = 30 | N = 30 |
| K = 5 | K = 3 | K = 6 |
| P = 1 | P = 2 | P = 1 |

Which of these three combinations to use depends on several factors:

1. PLLCON = 0x7D12 has the input divider P at 2 so that a clock source with a non-50/50 duty ratio such as an oscillator module may be used as the clock source.
2. 0x7884 uses a lower N multiplier and higher K divider than 0x7D12. This reduces the PLL jitter on the $f_{PLL}$ output compared with (i).
3. 0x7D85 gives the lowest clock jitter of all as K = 6.

Therefore (iii) is to be preferred where an external crystal is being used.



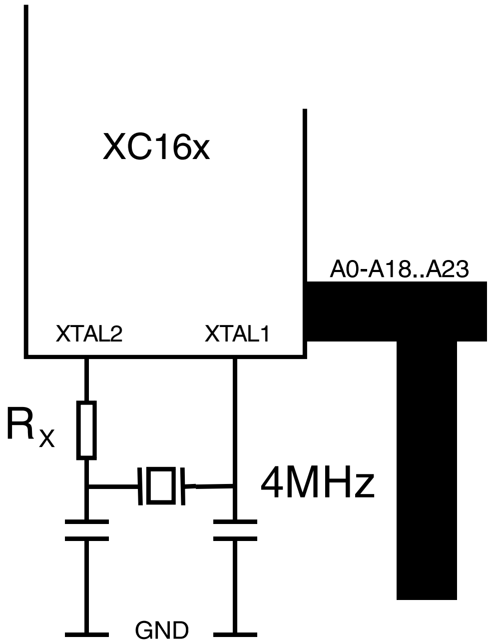

**Approximate Accumulated PLL Jitter**

**Bold lines indicate the minimum possible jitter. These can only be achieved by setting the maximum possible K for the combination of required $f_{MC}$ and crystal frequency.**

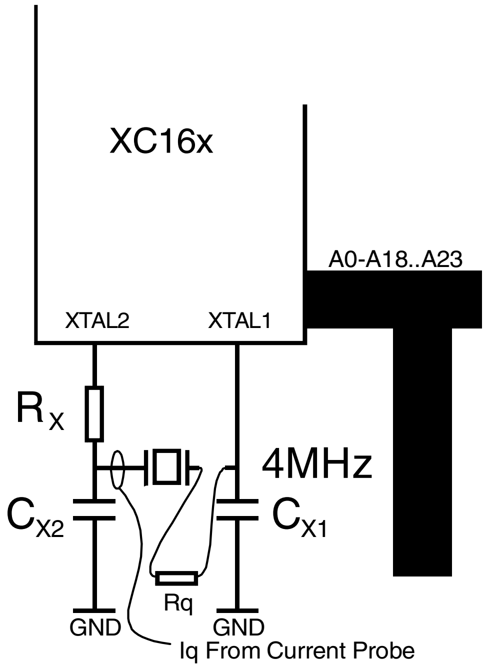
# 2.6 Generating The Clock

## 2.6.1 Designing Clock Circuits

There are two basic choices of clock source, the crystal or a self-contained oscillator module. The design of a traditional clock circuit is not a trivial task and requires some care to get reliable start-up when production tolerances and component ageing is taken into account. The XC166 is family is no more demanding in this area than any other microcontroller ,so the hints given in the following section should be considered for any clock circuit design.




**Fundamental Mode Operation**




**Test Configuration**

## 2.6.2 Oscillator Modules

Using an oscillator module is very simple, as the operating point calculations will have been taken care of by the manufacturer. The EMC emissions are also less as the metal case is always grounded and there will be a shorter signal path. The only critical factor is that the rise and fall time should be less than 5ns. There is a small price premium over the conventional crystal-plus-capacitors approach but this is not great. Indeed, it is only if the microcontroller is going to be used in a 25k+ per annum quantity that the extra cost of a module is going to become significant. The oscillator output should be connected to the XC166's XTAL1 pin.

### 2.6.3 Designing Crystal Oscillator Circuits

The traditional clock circuit usually comprises a parallel resonant fundamental crystal plus two capacitors and a resistor to limit the current through the resonant device. The XC166 uses crystals in the range of 4 to 16MHz.

The selection of the series resistor value Rx must be made so that the oscillator is guaranteed to start within 0.1ms to 5ms, even after mass production tolerances and ageing effects are taken into account. It must also be chosen to keep the power drive level of the crystal between typically 50uW to 800uW, although the device's datasheet should be consulted.

The process of defining $R_X$ and the "load capacitors", $C_{X1}$ and $C_{X2}$, is aimed at making sure that there is sufficient current flowing through crystal to drive the on-chip inverter that produces the oscillation. The crystal has a characteristic resistance known as the "equivalent series resistance" or "load resonant resistance", which is a combination of its typical resistance (R1typ) and residual capacitance ($C_{0typ}$), as stated by the manufacturer, plus reactive effects due to the oscillation and the load capacitors $C_{X1}$ and $C_{X2}$. This equivalent resistance is given by:

$R_L = R_{1typ} \times (1 + (C_{0typ}/C_L)^2)$

*Where:* $C_L = (C_{X1} \times C_{X2})/(C_{X1} + C_{X2}) + C_S$ *($C_S$ = the stray capacitance of clock circuit)*

During this Rx definition phase, a small value resistor, Rq, should be inserted in series with the crystal. The temporary resistor, Rq, must be increased until the oscillator does ***not*** start automatically when the 166 is powered up for different values of load capacitor. This value will be Rqmax. For ease of adjustment, an RF potentiometer can be used but you must bear in mind that this is RF engineering and the value of Rq so arrived at must be verified by replacing the potentiometer with an equivalent SMD or RF resistor and repeating the test. The ratio of Rqmax to the equivalent series resistance is the "Safety Factor" and is a measure of how much spare capacity there is in the circuit to overcome tolerance and ageing effects:

Safety Factor (SF) = $R_{qmax}/R_L$

A current probe should be used to measure the peak-to-peak current (Ipp), converted to drive power with:

$P_w = (I_{pp} \times I_{pp} \times R_L)/8$

The resulting relationships between safety factor and power drive versus load capacitor value should be plotted on graph paper. From both curves, a value of load capacitors that gives the best combination of safety factor and power consumption can be chosen.

### 2.6.4 Crystal Oscillator Components Test Procedure

1. Select a value for Rx
2. Fit load capacitors, CX1 and CX2 of the value given in the table
3. Adjust Rq until oscillation will not self-start in less than 5ms when the XC166 is powered-on. Record this resistance in a table, similar to that given below:

**Test Record For Rx =** 680R

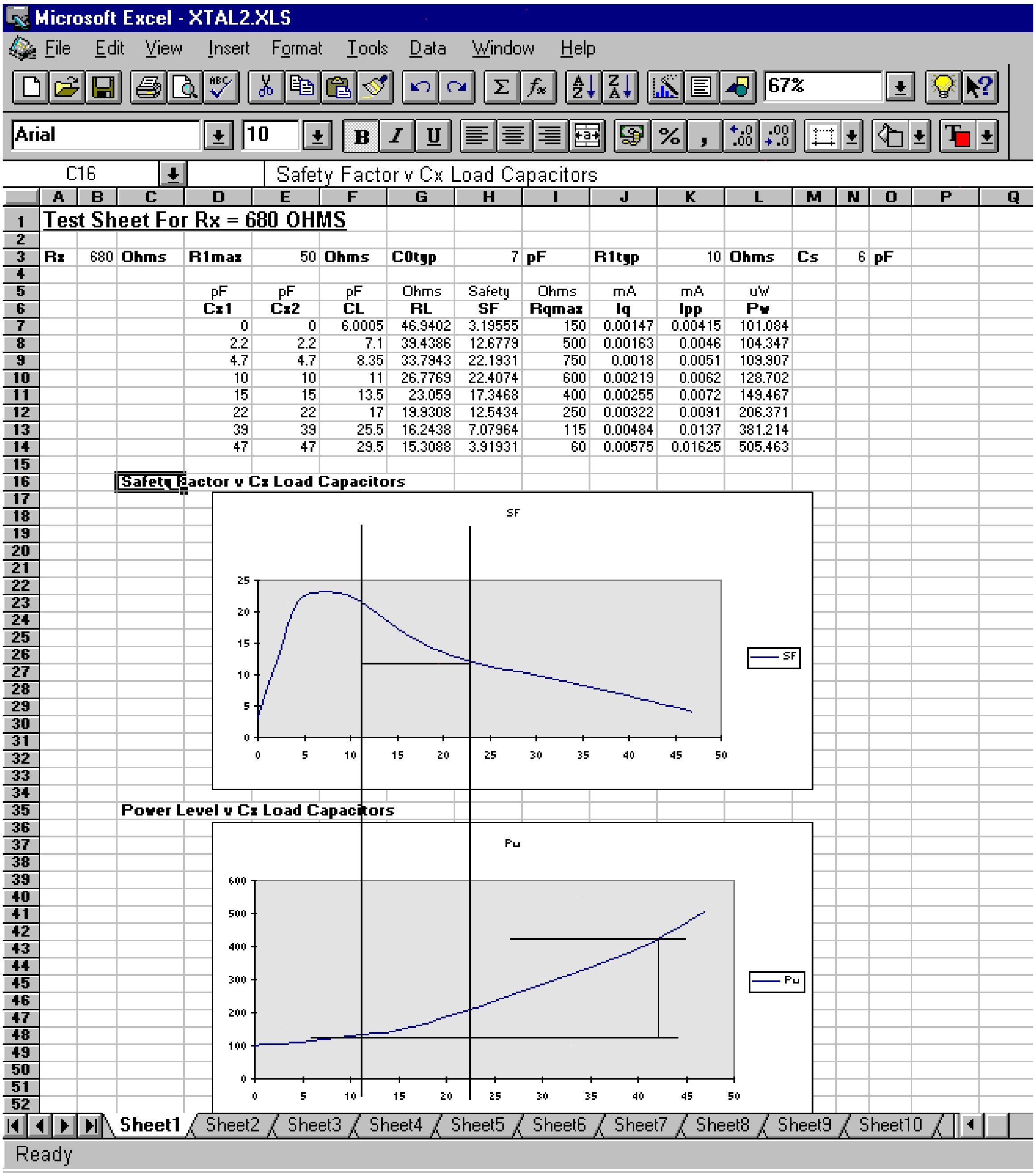
| CX1 = CX2 | Ipp | Pw | | Rqmax |
|---|---|---|---|---|
| 0.0pF | 0.002075 | 25.27 | | 150 |
| 2.2pF | 0.0023 | 26.09 | | 500 |
| 4.7pF | 0.00255 | 27.48 | | 750 |
| 10pF | 0.0031 | 32.17 | | 600 |
| 22pF | 0.00455 | 51.59 | | 250 |
| 47pF | 0.008 | 122.5 | | 60 |

4. Select the next value of load capacitors and repeat steps 2 to 4
5. Now pick another value for Rx and repeat the procedure.

After a number of Rx values have been tested in this way, the resulting curves should be examined for the resistor and load capacitor values that give the best safety factor at a power level of 50uW-800uW. Having selected the values, the resistor Rq should be removed and the current and start-up times rechecked.

If an adequate safety factor cannot be achieved, particularly above 20MHz, it is possible to add a series 1-10M resistor to increase the feedback to the XTAL1 input pin. Otherwise, a third-overtone mode must be used. Unfortunately, the component selection process is more complex and when it is considered that an extra inductor and capacitor will be required to damp-out the fundamental frequency, it might prove more cost-effective to use an oscillator module!

To simplify the selection process, Infineon can provide an Excel spreadsheet template at [www.infineon.com/xc166-family](http://www.infineon.com/xc166-family) that automates the conversion of test results and characteristic curve plotting, as




| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Rx | 680 | Ohms | R1max | 50 | Ohms | C0typ | 7 | pF | R1typ | 10 | Ohms | Cs 6 pF |
| | | | pF | pF | pF | Ohms | Safety | Ohms | mA | mA | uW | |
| | | | Cx1 | Cx2 | CL | RL | SF | Rqmax | Iq | Ipp | Pw | |
| | | | 0 | 0 | 6.0005 | 46.9402 | 3.19555 | 150 | 0.00147 | 0.00415 | 101.084 | |
| | | | 2.2 | 2.2 | 7.1 | 39.4386 | 12.6779 | 500 | 0.00163 | 0.0046 | 104.347 | |
| | | | 4.7 | 4.7 | 8.35 | 33.7943 | 22.1931 | 750 | 0.0018 | 0.0051 | 109.907 | |
| | | | 10 | 10 | 11 | 26.7769 | 22.4074 | 600 | 0.00219 | 0.0062 | 128.702 | |
| | | | 15 | 15 | 13.5 | 23.059 | 17.3468 | 400 | 0.00255 | 0.0072 | 149.467 | |
| | | | 22 | 22 | 17 | 19.9308 | 12.5434 | 250 | 0.00322 | 0.0091 | 206.371 | |
| | | | 39 | 39 | 25.5 | 16.2438 | 7.07964 | 115 | 0.00484 | 0.0137 | 381.214 | |
| | | | 47 | 47 | 29.5 | 15.3088 | 3.91931 | 60 | 0.00575 | 0.01625 | 505.463 | |

**Excel Spreadsheet For Oscillator Component Evaluation**

illustrated below:

The clock circuit will in fact produce two drive currents: until the RESOUT pin goes high, (after the EINIT instruction is executed), the current drive of the clock will be greater to ensure that the CPU is less likely to be upset during the potentially noisy initialisation phase. It also helps to overcome the initial high resistance of the crystal during the startup phase.

Typical load capacitor values are 22pF with Rx around 1K. However, you should not rely on these and for any serious project, the selection procedure given earlier should be followed.

### 2.6.4.1 Typical Component Values

The table gives typical values for a selection of commercially available crystals. These must not be used as they stand without testing - we deliberately have not given the brand names for this reason! It is recommended that you compare the characteristics C0typ, R1typ (in the shaded panels) and fundamental frequency of your device with the examples in the table and pick the one which is closest.

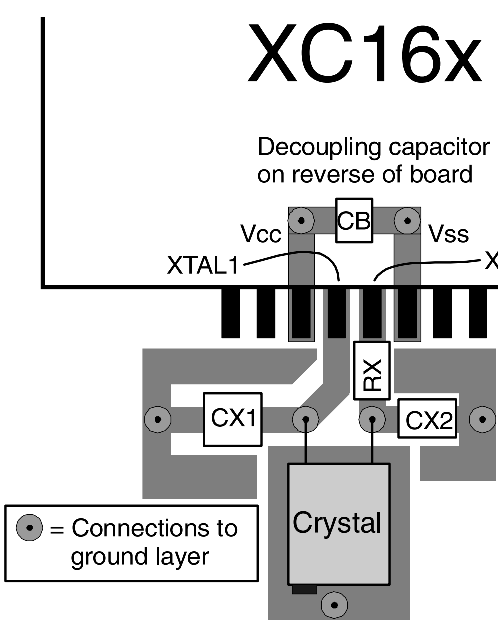
Make up a clock circuit using the load capacitors CX1 and CX2 plus the series resistor Rx and perform the check of safety factor and drive power given in the previous section. The chances are that the results will be within limits but it would be very embarrassing if reliability problems occur in production and you have to admit that you never verified the component values in the clock circuit....

| Frequency (MHz) | Rx2 (Ohm) | CX1 (pF) | CX2 (pF) | CL (pF) | C0typ (pF) | R1typ (Ohm) | R1max (Ohm) | R1max (TK) (Ohm) | Pw (uW) | Rqmax (Ohm) | Safety Factor (SF) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 40 | 0 | 12 | 15 | 13 | 7 | 10 | 50 | 60 | 420 | 300 | 2.11 |
| 32 | 0 | 12 | 15 | 11 | 5 | 15 | 50 | 60 | 520 | 390 | 3.07 |
| 24 | 180 | 15 | 22 | 12 | 5 | 15 | 50 | 60 | 510 | 390 | 3.24 |
| 20 | 390 | 8.2 | 39 | 10 | 4 | 20 | 60 | 80 | 375 | 560 | 3.57 |
| 18 | 390 | 12 | 39 | 14 | 4 | 20 | 60 | 80 | 335 | 540 | 4.08 |
| 16 | 390 | 12 | 47 | 13 | 4 | 20 | 60 | 80 | 353 | 580 | 4.24 |
| 12 | 390 | 15 | 47 | 13 | 4 | 30 | 70 | 90 | 312 | 1000 | 6.50 |
| 10 | 390 | 15 | 47 | 14 | 3 | 30 | 80 | 100 | 216 | 1200 | 8.14 |
| 8 | 390 | 15 | 47 | 15 | 3 | 35 | 80 | 100 | 372 | 1800 | 12.50 |
| 6 | 390 | 15 | 47 | 14 | 3 | 35 | 80 | 140 | 100 | 2200 | 10.66 |
| 5 | 390 | 22 | 47 | 18 | 3 | 35 | 80 | 140 | 110 | 2700 | 14.17 |
| 4 | 390 | 22 | 47 | 16 | 4 | 20 | 80 | 150 | 46 | 3300 | 14.08 |

**Typical Crystal Characteristics And Component Values**

### 2.6.5 Laying Out Clock Circuits

The layout of the clock circuit can be critical in determining both the RF emissions and susceptibility of a XC166 design. As with any high frequency system, the loop areas must kept as small as possible, meaning in practice that all components must be located as close as is practicable to each other and to the XTAL1/XTAL2 pins on the CPU. With metal-canned crystals, the case should be soldered to a grounded area on the top surface plus it should be connected to the main ground layer in a multi-layer board. This will also improve the mechanical stability of the part.




**Sample Oscillator Circuit Layout**

Inductive and capacitive coupling can be reduced by eliminating parallel runs of tracks either on the same layer or between layers. The grounding of the load capacitors should have a generous track width and be connected directly to the ground layer to avoid ground loops, which are a major source of RF emissions.

### 2.6.6 Symptoms Of A Poor Clock

It must be emphasised that the series resistor value must be chosen with care. An incorrect value is unlikely to result in a total CPU failure, or even in erratic operation of the core, timers or A/D converter. However, the first symptom of a poor choice is that an unexpectedly large number of bus errors on the CAN peripheral may be seen, or the ALE timing is erratic for no readily apparent reason. Such behaviour should ***never*** be ignored - try shorting the resistor out to see if the problem goes away....

## 2.7 Real Time Clock Oscillator

A Real Time Clock (RTC) is available on the XC161 and XC167 that can optionally have its own clock crystal. The main oscillator (divided by 8) can be used as the input and will still allow the real time clock operation to be continued during XC166 sleep mode. For maximum time-keeping accuracy though, an external 32.768kHz "watch" crystal needs to be provided on the X3 & X4 pins. Before entering sleep mode the RTCCM bit in SYSCON0 must be set to 1 to enter the "asynchronous" mode, provided $F_{CPU}$ is greater than 4 x $F_{count}$. However as this dedicated oscillator is not synchronised to the CPU clock, the time registers cannot be read or written. Therefore after a system reset, the operating mode must be set back to synchronous (RTCCM = 0). The current consumption during sleep mode with the RTC running is around 100uA.

A complete powering down of the device that requires a power-on reset to restart will leave the real time clock registers undefined i.e. any accumulated time will be lost.

## 2.8 Further Information On Oscillator Design

Application notes AP24205 and AP242401 available from www.infineon.com/xc166-family that cover crystal oscillator and ceramic resonator design in detail.

# 3 Bus Modes And Timings

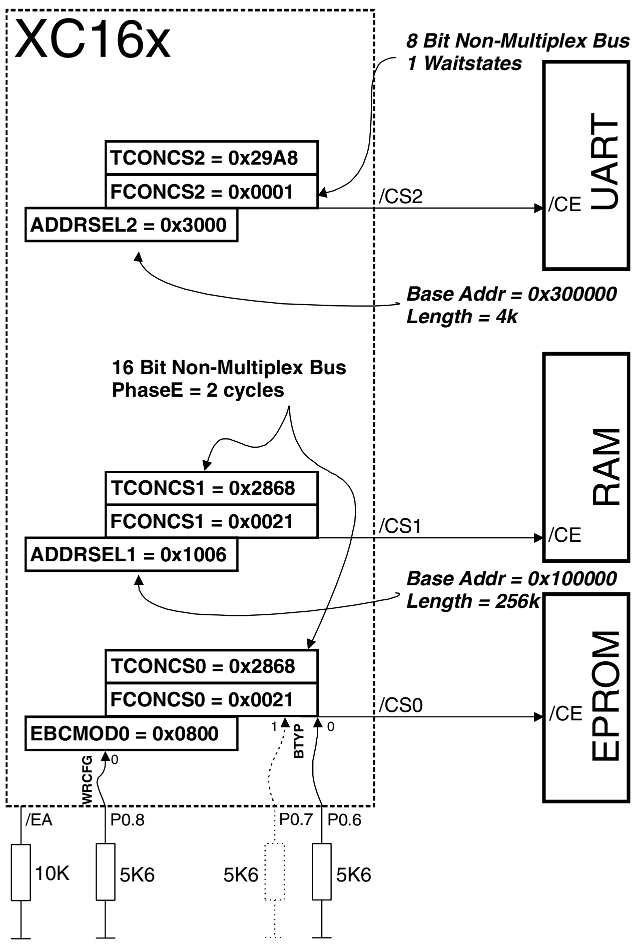
## 3.1 Flexible Bus Interface

The XC166 family is primarily intended for single chip applications with variants having up to 256k of automotive on-chip flash and up to 12k of RAM. However, the provision has been made for a very flexible external bus interface. The basic philosophy behind the XC's bus interface is simplicity, even in external bus systems; by providing 8- and 16-bit non-multiplexed modes, it is possible to dispense with an address latch and provide just a FLASH and RAM to make a working system. The integral software-programmable chip selects can make most address decoder logic redundant. Thus, despite its 20 fold improvement in performance, an XC's digital design can be simpler than that of an 8031!

If an external bus is required, one of the XC's most useful features is its ability to support two different bus configurations in a single hardware design. Thus whilst the external FLASH and RAM areas (if needed) can be 16-bit non-multiplexed with zero waitstates for best speed, slow (and low cost) peripherals such as RTCs can be addressed with, for example, an 8-bit bus with 3 waitstates.

### 3.1.1 Integral Chip Selects

The XC family can have up to 5 external chip select regions. Chip selects 1 to 4 each have ADDRSEL, FCON and TCON registers that individually control the bus mode, timing and range of the Chip Select region. CS0 doesn't have an ADDRSEL register and is active for any addresses that aren't covered by another Chip Select (or the internal memory / registers). While looking at the user manual it may seem that CS 5,6 & 7 are also present, but in reality these are used to address internal LX bus peripherals. CS 5 & 6 are reserved for future use and CS7 is for the TwinCAN Module and is located at 0x200000.



| Chip Select | Pin Name | Control Register | Address Range Register |
|---|---|---|---|
| /CS0 | P6.0 | TCONCS0, FCONCS0 | Not Applicable |
| /CS1 | P6.1 | TCONCS1, FCONCS1 | ADDRSEL1 |
| /CS2 | P6.2 | TCONCS2, FCONCS2 | ADDRSEL2 |
| /CS3 | P6.3 | TCONCS3, FCONCS3 | ADDRSEL3 |
| /CS4 | P6.4 | TCONCS4, FCONCS4 | ADDRSEL4 |

It is essential when setting up the ADDRSEL, FCON and TCON registers to make sure that you configure the ADDRESELx before the corresponding FCONCSx and TCONCSx. If you do not, the CPU will enable the ADDRSEL for an undefined bus configuration and a crash will ensue! Also note that while you may initialise these registers from C, any variables located in an region controlled by them will not be zeroed before `main()` by the compiler start-up code as the corresponding chip select will not be activated (low).

#### 3.1.1.1 Overlapping Chip Selects

It is possible to overlap the regions covered by external chip selects, provided that certain rules are observed. When an address is generated that is to be accessed, the CPU decides which device on the bus receives the address according to the following rules:

First the CPU checks whether the address corresponds to an on-chip memory region or peripheral and directs the access internally so that it will not appear on the external bus. Thus for example, external memory areas that partially overlap internal resources will be inaccessible.

Next registers ADDRSEL2 & 4 are checked to see what areas they cover. A match with one of these registers directs the access to the respective external area. Thus ADDRSEL 2 & 4 regions can overlap ADDRSEL1, 2 & 7 but not each other. The overlapping of windows of ADDRSEL2 & 4 gives undefined behaviour.

Next ADDRSEL1, 3 & 7 are checked and if the address lies in an area covered by one of these. Again, ADDRSEL1, 3 & 7 may not overlap. However ADDRSEL2 & 1 may overlap each other, as can 4 & 3. These pairings are the only ones that are allowed.

Finally, addresses that are not in the range of any other ADDRSEL register cause chip select 0 to be used. Thus chip select 0 can overlap all other chip selects.

## 3.2 Setting The Bus Mode

### 3.2.1 On-Chip Boot

If the /EA pin is pulled high during reset, the processor will boot from the internal flash and the external bus controller will be disabled. If required, the external bus can be configured through software. In single chip mode virtually all of the signals used by an external bus are available for use as I/O. Two External Bus Controller Mode Registers (EBCMOD 0 and 1) configure things such as the number of CS signals and segment address lines required, enabling the ALE and /BHE signals as well as selecting ports 1 and 2 as address and data busses.

### 3.2.2 External Boot

With /EA low during reset, the XC reads the pattern of user-defined pull-down resistors on the P0.6 and P0.7 to set the default bus mode. In fact, the pull-down resistor pattern is placed into the BTYP field in the FCON0 register where it can be changed by software, although it is definitely not recommended to do this on external ROM designs. The number of chip selects and the overall address range of the processor are also set via Port 0 pull-down resistors, covered in section 2. These can also be changed in software by modifying the EBCMODx registers but this should not be necessary.

## 3.3 Setting The Overall Addressing Capabilities

By enabling up to 8 segment address lines over and above the 16 lines of Port 0 as segment address lines (in demultiplexed bus mode), a 16MB memory device can be used. Assuming the processor's chip select signals are going to be used, the number of segment address lines required is determined by the largest device to be connected. For example, if a 256KB memory was connected to CS1 and an Ethernet controller (with a much smaller address range) was connected to CS2, only two segment address lines would need to be enabled. By setting the ADDRSEL1 and 2 registers the devices could be enabled (pretty much) anywhere in the 16M address space.

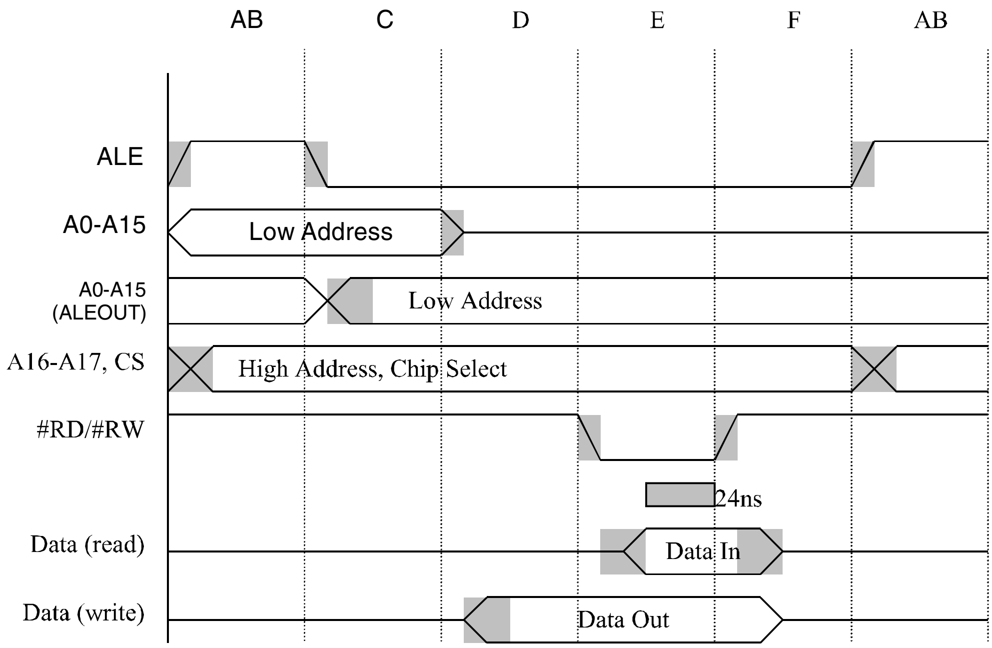
### 3.3.1 External Memory Access Times

The bus access mode used by the XC166 is Intel-style. The timing for each of the processor's 5 chip select regions is configured by its TCONx register. A bus cycle is divided into six different timing phases and the number of clock cycles for each of these phases is set in the TCONx register. The phases are:

| | | |
|---|---|---|
| **A Phase** | **/CS Changing phase** | **0 – 3 clock cycles** |
| **B Phase** | **Address Setup /ALE Phase** | **1 – 2 clock cycles** |
| **C Phase** | **Delay Phase** | **0 – 3 clock cycles** |
| **D Phase** | **Write Data Setup / MUX Tristate Phase** | **0 – 1 clock cycles** |
| **E Phase** | **RD/WR Command Phase** | **1 – 32 clock cycles** |
| **F Phase** | **Address / Write Data Hold Phase** | **0 – 3 clock cycles** |

This allows a great deal of control over a bus cycle to cater for different characteristics of external devices. In most applications, Phase A will be equal to 0. If the TWINCAN module is being used, it will always be 0.

### 3.3.2 Calculating The Bus Timing Parameters For A Multiplexed Bus




**Bus Timing Example (multiplexed)**

The bus timings are based on devices used on Infineon starter kits. In this example, Samsung K6R4016C1D SRAM is fitted. This memory device is organised as 256K x 16 and have an access time of 12ns. The bus is multiplexed.

The bus timings will be calculated with a 40MHz CPU clock. The timing is always designed for the worst-case manufacturing tolerances. Please note that the XC-microcontroller's signals do not always appear in sync. at the output as the rise/fall time is up to 4ns. The delay on the falling edges of the CS, RD and WR-signals can amount to up to 13ns. In the case of rising edges, this can be up to 6ns.
At 40MHz, one clock has a duration of 25ns.

**Phase A: 0 Clocks**
Phase A is only necessary when working with more than one chip select and when it is not certain if the bus is still being driven from a previous write- or read access. Phase A clocks will only be inserted when the address being accessed causes a change of chip select, such as might happen when code being fetched from external FLASH accesses data in external RAM. Most XC166 designs use on-chip FLASH and the only external access might be to an external SRAM, so no chip select changes will occur. Here it is assumed that we are only working with 1 chip select and have set the value to 0, as in fact is almost always the case.

**Phase B: 1 Clock**
This clock is required in order for the address latch to capture the data. As a rule, the address latch is not critical at high clock speeds. However please note that the guaranteed high-time only amounts to 2ns.

**Phase C: 1 Clock**
Phase C serves to delay and through this, the address latch is able to accept the data. In a worst-case scenario, the ALE can still be 10ns long. High < 1 Clock.

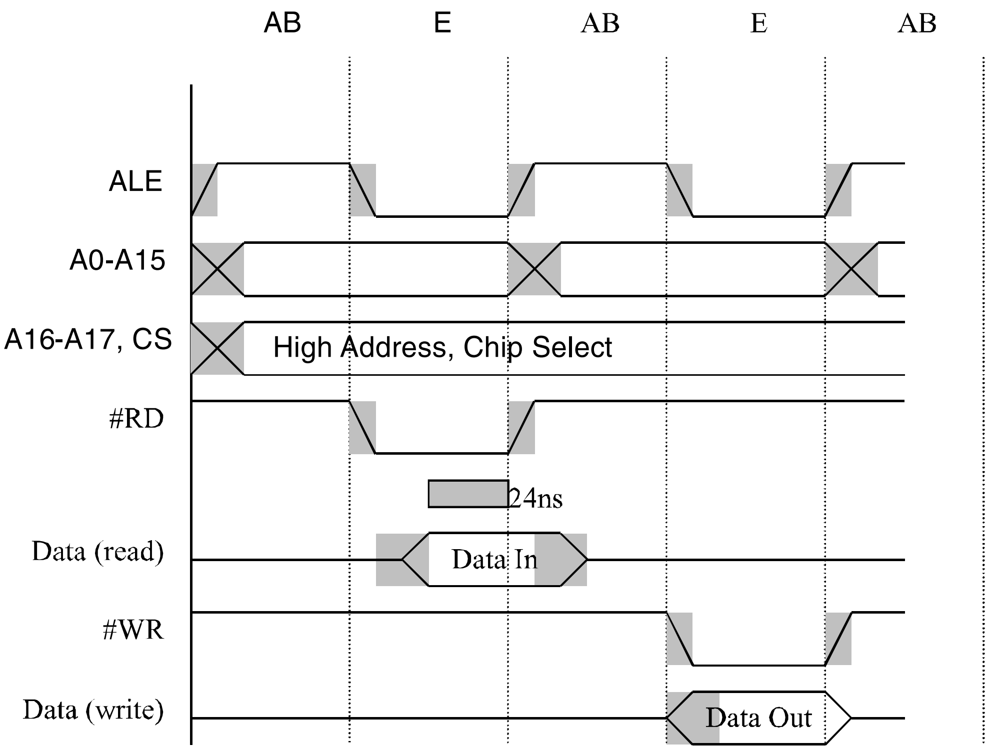
**Phase D: 1 Clock**
The microcontroller puts address lines into a tristate state in order to prevent the XC-microcontroller from continuing to drive them, as well as simultaneously lining up the data from the memory. Max duration: 13ns < 1 Clock.

**Phase E: 2 Clock**
After activation, the READ line lasts up to 13ns until this is in the low state. The READ signal's rising edge may only be completed when the data on the bus has been there for at least 24ns.
Under worst-case conditions, this will result in a time of 13ns + 6ns + 24ns = 43ns. This corresponds to 2 Clocks.

**Phase F: 1 Clock**
After deactivation of the READ line it takes up to 6ns until it reaches the high state. Then it will take a further 6ns until the memory has set all lines into the tristate condition. 6ns+6ns = 12ns < 1 Clock.

**Note:** The values for the memory were taken from Samsung data. The values for the XC-Microcontroller were taken from an Infineon data sheet.

### 3.3.3 Calculating The Bus Timing For A Demultiplexed Bus




**Bus Timing Example (Demultiplexed)**

The bus timings are based on devices used used on Infineon starter kits. As is the previous example, Samsung K6R4016C1D SRAM was employed. This memory device is organised as 256k x 16 and has an access time of 12ns. The bus is demultiplexed.

The bus timing will be calculated with a 40MHz CPU clock. The data given by the manufacturer are worst-case timings. Please note that the XC microcontroller's signals do not always appear simultaneously at the output, as the rise/fall time of the edges can be up to 4ns. Delays in the CS, RD and WR signals' falling edges can be up to 10ns. In the case of rising edges, it can be up to 6ns.

At 40MHz one clock has a duration of 25 ns.

**Phase A: 0 Clocks**
Phase A is only necessary when working with more than one chip select and when it is not certain if the bus is still being driven from a previous write- or read access. Phase A clocks will only be inserted when the address being accessed causes a change of chip select, such as might happen when code being fetched from external FLASH accesses data in external RAM. Most XC166 designs use on-chip FLASH and the only external access might be to an external SRAM so no chip select changes will occur. Here it is assumed that we are only working with just 1 chip select and set the value to 0.

**Phase B: 1 Clock**
This clock is always available and cannot be disabled by configuration.

**Phase C: 0 Clocks**
This phase does not apply with the demultiplexed bus.

**Phase D: 0 Clocks**
This phase does not apply with the demultiplexed bus.

**Phase E: 2 Clocks**

After activation of the READ line, it takes up to 13ns to go fully low. The data is then inserted onto the bus within 6ns by the memory. The READ signal's rising edge may only be completed when the data on the bus has been present for at least 24ns.

Under worst-case conditions, this will result in a time of Es = 13ns + 6ns + 24ns = 43ns. This corresponds to 2 clocks.

**Phase F: 0 Clocks**

This phase is not present with a demultiplexed memory interface.

Where external memory accesses are at 40MHz, 3 clocks are always required.

Finally we now need to reverse the calculation to see how fast the CPU can be clocked in order to get bus accesses to take place in 2 clocks.

The timing of Phase E is employed.

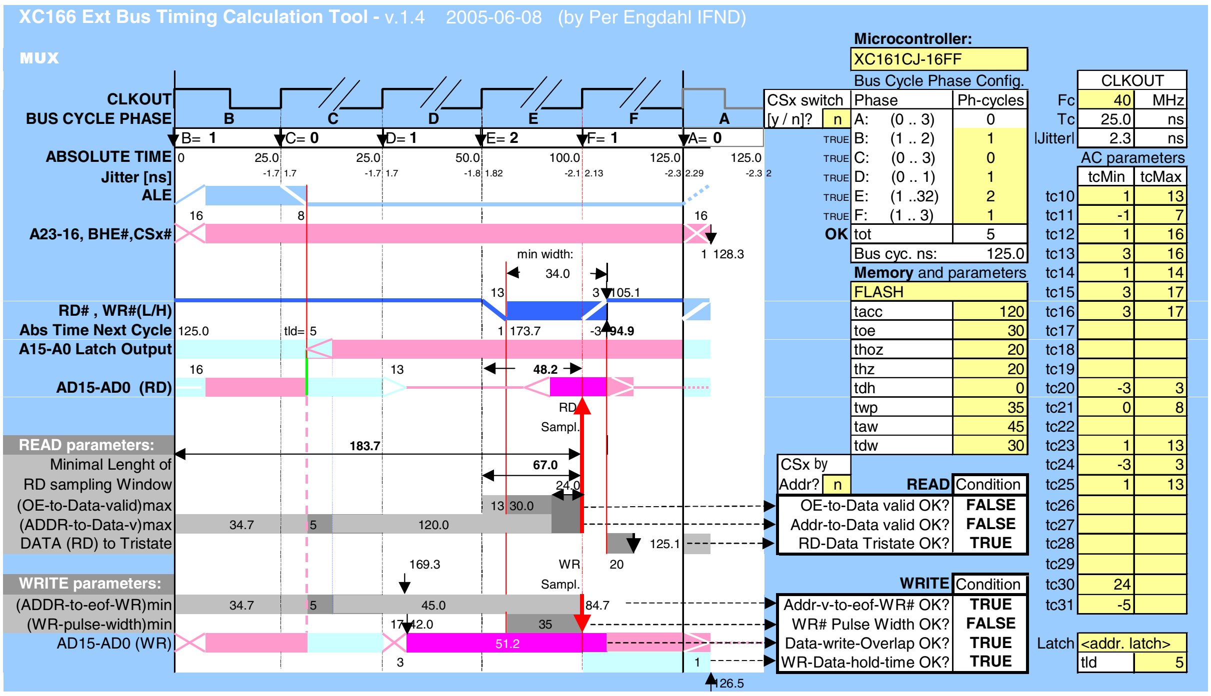
$$f_{\max} = \frac{1}{t_E}$$

$$f_{\max} = \frac{1}{43ns} = 23{,}25MHz$$

If the program is implemented from the internal memory, it may not be wise to use this value. It may be better to run the program at 40MHz and access the memory with longer phases.

## 3.3.4 A Tool For Calculating The Bus Timing Parameters

A spreadsheet and application note (AP1608810) are available from Infineon that allows the values of the various phases to be calculated automatically, given the memory access time of the memory device and the CPU clock speed. These can be found at [www.infineon.com/xc166-family.](http://www.infineon.com/xc166-family)




**Bus Timing Calculator Spreadsheet From Infineon**

# 4 Interfacing To External Memory Devices

Despite the power of the XC166 architecture, the additional hardware necessary to get a device up and running is very small. In many applications the large on-chip RAM and FLASH are sufficient. In some cases, an external RAM is added, especially if the application uses a lot of data. In large systems, the XC166 will boot into the internal FLASH and any time-critical program sections will be in this area. Other code and constant data will be in external FLASH. It is possible for the XC166 to boot from external FLASH but it really makes almost no sense to do so. Settings for bus width (8/16-bit) and type (mux/demux) are best made via the RSTCFG register from software running from the internal FLASH, rather than from using pull-down resistors on Port 0 in classic C167-fashion!

The standard XC166 series has a 5v external bus, so you must ensure that any memory device chosen is compatible with this. Future versions will support 5v or 3.3v busses, giving a wider choice of devices, especially if lead-free ROHS-compliance is an issue.

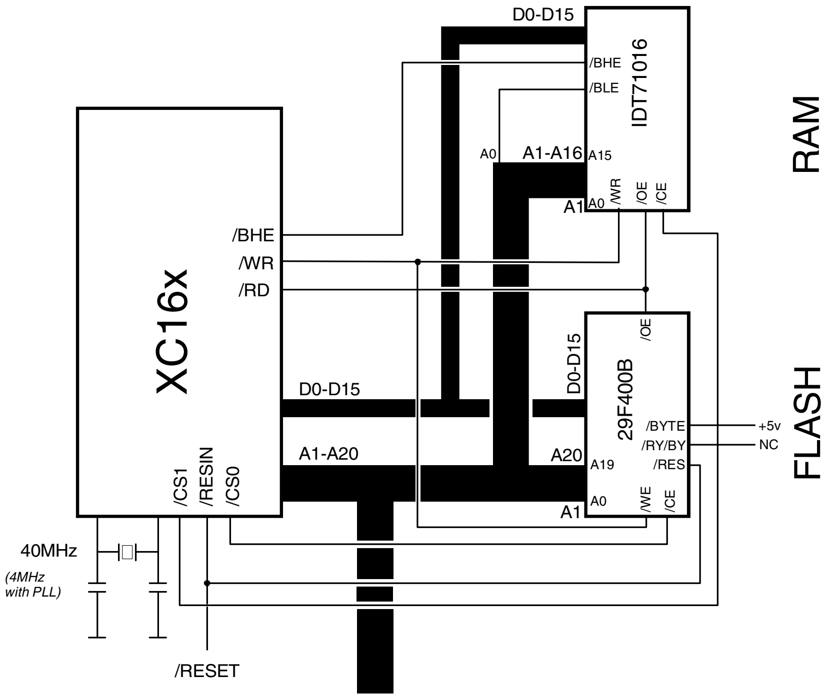
The following diagrams in the next sections illustrate simple examples of common configurations.

## 4.1 Using 16-Bit Memory Devices

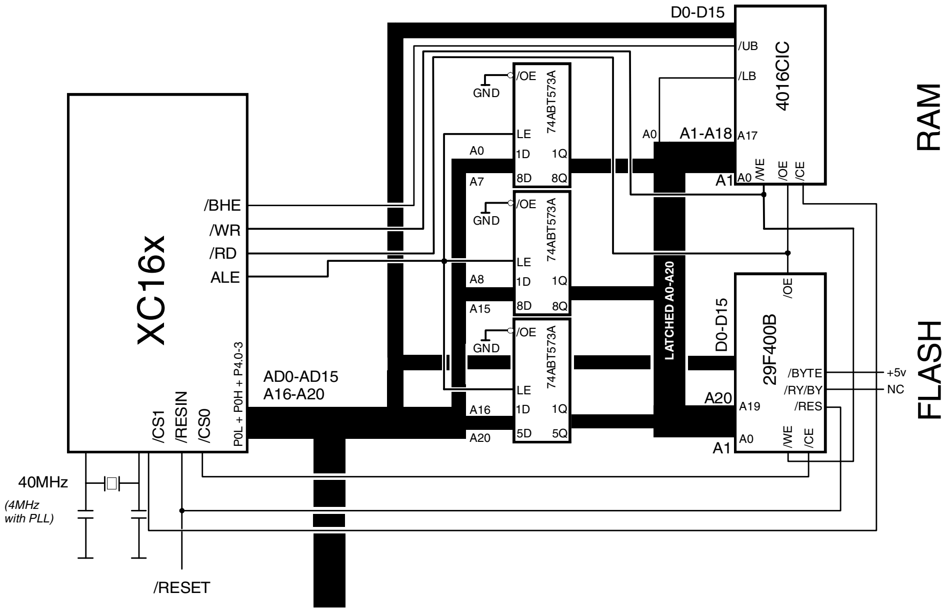
This is now the most common way to expand the memory in an XC166 design. 16-bit FLASH and RAM is now comparatively cheap. Adding a 16-bit demultiplexed bus will use up the whole of Ports 1 and 2 plus up to 8 bits of Port 4, which represents a significant loss of IO. However it does give the fastest access times for code and data. It should be noted, though, that code execution from external ROM is much slower than from the internal FLASH. This is mainly due to the fact that the internal bus is 64 bits wide, whereas the external bus is just 16 bits. As 16-bit memories are word-orientated, the XC166's A0 in fact is not really used as an address line and it is A1 that should be routed to the memory device's A0.

On smaller XC166 versions, it might be necessary to run the external bus in 16-bit multiplexed mode. This frees up Port 1 completely but the access times are approximately doubled, with code execution speed worst affected. Therefore this should only be done where large external data RAMs need to be added.

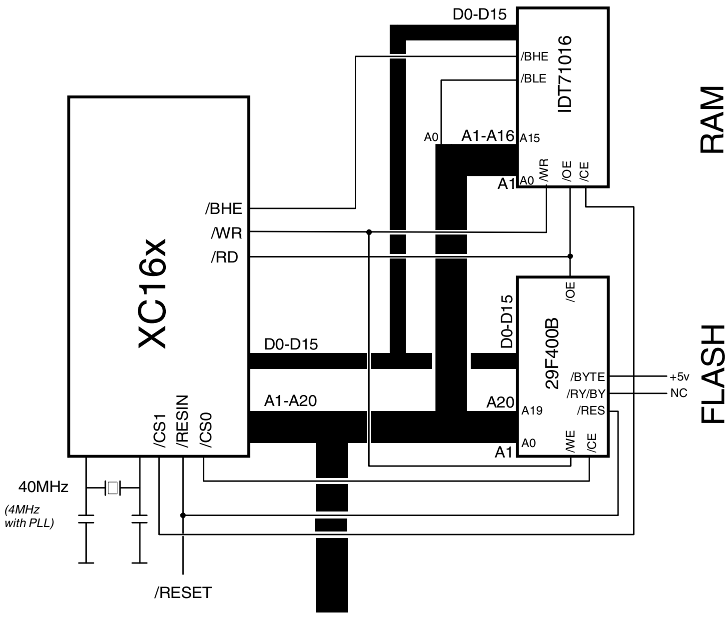
In these examples, FLASH programming routines must be word-wise as byte writes are not supported, although the XC166 itself does support this. This is unlikely to be a problem where the FLASH contains static code and constant data. Where the FLASH will also contain some adaptive data that is updated as the system runs, such variables must of types int or long, to eliminate the possibility of byte-writes.



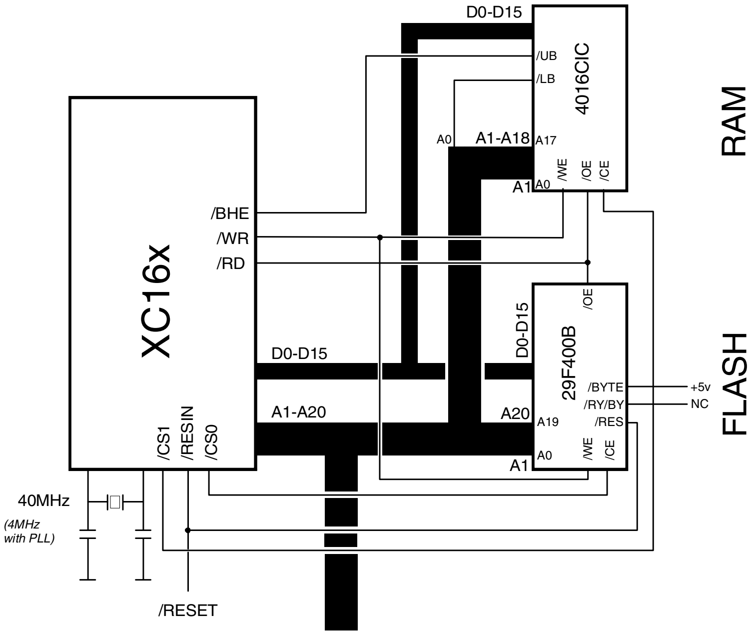

**16-Bit IDT71016 – Demultiplexed Bus**




**16-Bit Memories – Multiplexed Bus**




**16-Bit IDT71016 – Demultiplexed Bus**

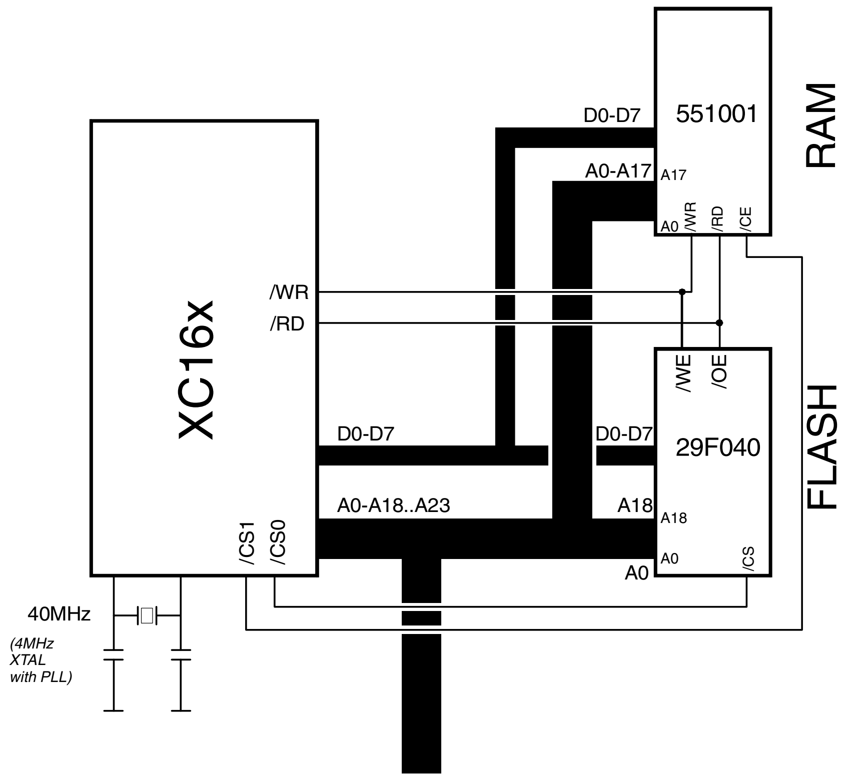



**16-Bit Memories – Demultiplexed Bus**

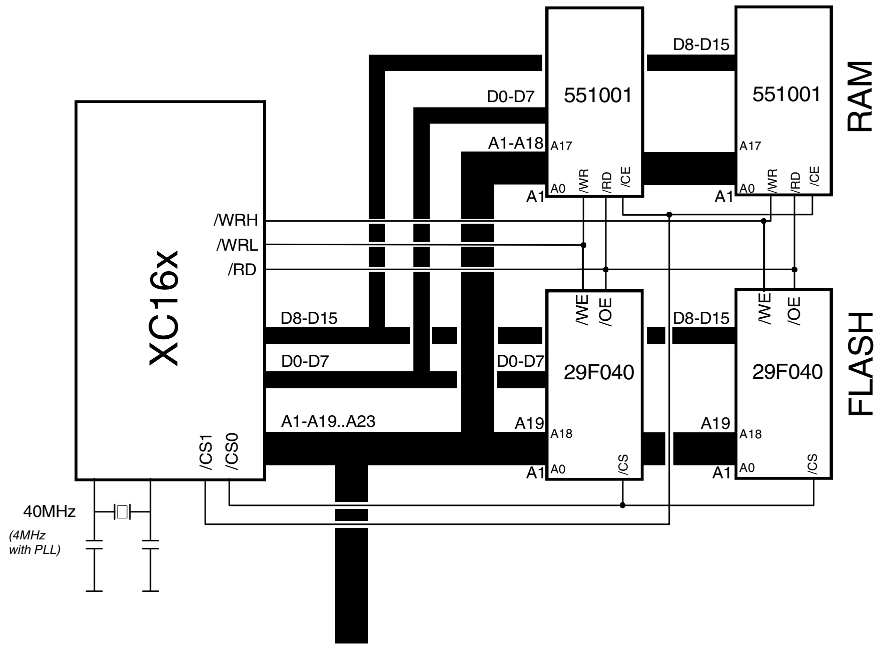
## 4.2 Using Byte-Wide Memory Devices In 16-bit XC166 Systems

To successfully use 8-bit RAMs the user must remember that the A0 pins on the RAMs go to A1 on the XC166. One RAM has its data lines connected to the XC166's D0-D7 (LOW) and the other is wired to D8-D15. A0 is effectively redundant in such a configuration. Failure to realise this before committing to a PCB will result in a lot of track cutting and hand-wiring. When the CPU reads a word both RAMs are enabled simultaneously by /READ so that the CPU can read D0-D15 in one access across the bus. As the XC166 is a 16-bit machine, all read accesses are word-wide, even byte ones - the unwanted byte is simply discarded.

For writes to RAM some means of only enabling one of the RAM's is required, as a byte write to an even location would corrupt the associated odd byte. The traditional method of preventing this is to create individual /WRITE signals for each RAM from /BHE and A0. However the XC166 has special /WRITEHIGH (/WRH) and /WRITELOW (/WRL) pins, which are connected to the corresponding /WR pins on the high and low RAMs. To enable this feature, the user must write a '1' into the EBCMOD0 register, bit 10. Alternatively, if external start mode is being used (/EA = 0), a pull-down resistor must be fitted to P0L bit 0 (P0.8).



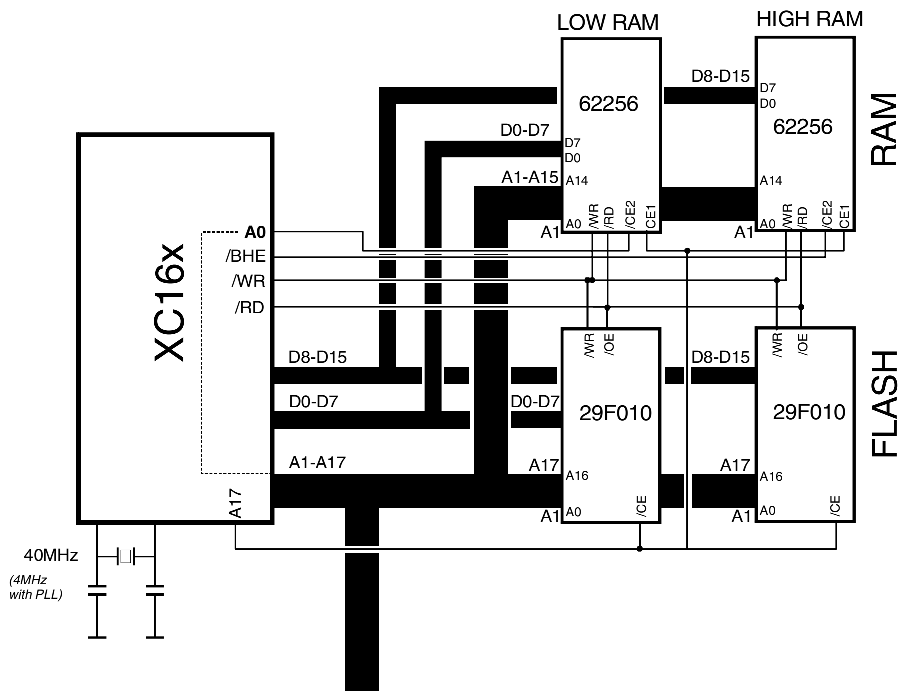

**8-Bit Memories – Demultiplexed 8-Bit Bus**




**8-Bit Memories – Demultiplexed 16-Bit Bus Using WRH/WRL Mode**

## 4.3 Using The XC166 With Byte-Wide Memories And #BHE

Where the WRITEHIGH (/WRH) or WRITELOW (/WRL) signals are not being used, a different approach is required. If 8-bit memory devices are chosen that have two chip selects available, then the /BHE (BYTE HIGH ENABLE) and A0 lines can be used to enable either the high or low bank of memories.




**8-Bit Memories – Demultiplexed 16-Bit Bus Using WR/BHE Mode**

Here the A0 and /BHE signals are connected to the active low chip selects on both RAMs. When an even byte is addressed the A0 is low and /BHE is high, so that the low RAM is enabled. On addressing an odd byte, A0 is high and /BHE is low, so that the high RAM is enabled.

A17 goes to the active high chip select, so that the RAMs are enabled above 0x20000. It also goes to the active low ROM chip select, mapping it to address zero.

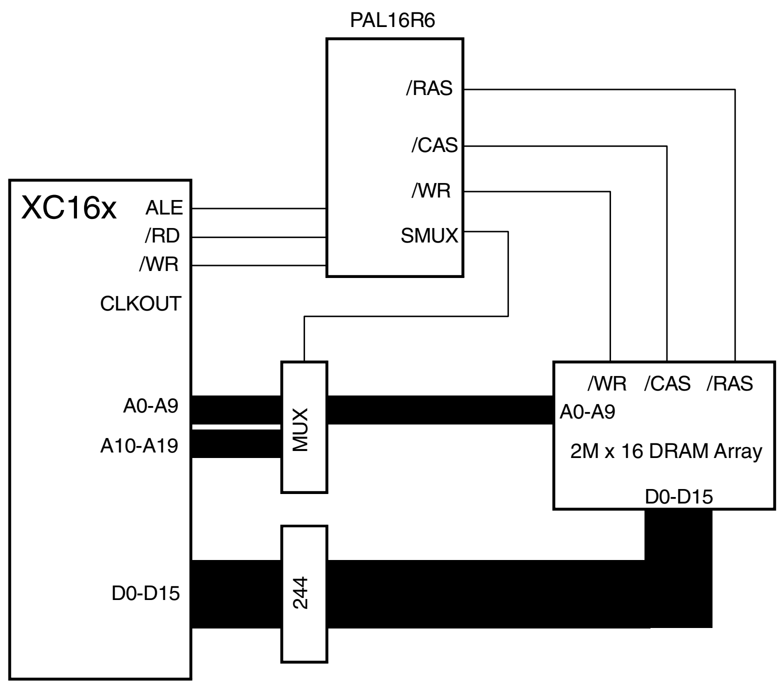
## 4.4 Using DRAM With The XC166 Family

Despite the recent falls in the cost of large, fast static RAMs, the PC-style SIMM DRAM is still the cheapest means of getting a very large RAM area in a XC166 design. In conventional CPU designs some method must be provided to refresh the memory. This is typically a bus request every few tens of microseconds and performing a RAS (Row Address Strobe) cycle only. A row address counter would then be incremented. The hardware for this requires an additional bus master with its own buffers and a counter.

Another commonplace refresh method is the CAS (Column Address Strobe) before RAS, which uses an internal row address refresh counter but still requires complex logic to ensure that precharge times are met. The XC166 family can perform the refresh task with virtually no external logic - all that is needed is a GAL to implement the RAS and CAS timing for accessing the DRAM.

The important peripheral is the PEC (Peripheral Event Controller), which is covered in detail in section 8.4. In the current context it is simply used (in conjunction with an on-chip timer) as a means of generating a read from each row every 15.6us. The PEC source pointer is used to make the row read and then automatically incremented to the next row. The destination simply throws away the read data by writing it to an unused location in the on-chip SFR area. The period is calculated as the refresh time divided by the number of rows in the DRAM. In a typical 256k x 4 HYB14256 DRAM this would be 8ms/512 rows = 15.6us.

As the PEC steals one 40ns cycle for every transfer, the % CPU overhead for the DRAM refresh is 40 x (0.1/15.6), which is negligible.




**DRAM Refresh With The XC16x**

## 4.5 Using FLASH Memory Cards With The XC166

### 4.5.1 Cheap Gigabyte Storage

It is becoming increasingly common for microcontroller applications to need very large non-volatile storage. This might be for storing values as part of a data logger or monitoring function or even for allowing the updating of the on-chip FLASH without using the serial port or bootstrap mode. Low-cost mass storage devices like Compact FLASH and SD/Multimedia cards are now widely available as a result of the popularity of digital cameras and PDAs. They hold out the prospect of vast memory capacity and easy movement of data between a PC and a microcontroller. Not only can data be moved to and from an embedded device, but even complete microcontroller application software updates are feasible just through a card insertion. Typical FLASH card storage costs are around 25 cents per MB and falling, so they are particularly attractive for memory-hungry applications like data-logging.

### 4.5.2 Using CompactFLASH Cards For XC166 Program Updates

An interesting use of FLASH cards is for program updates in the field. If the XC166 contains a boot program some applications use it to access a larger program in a Compact FLASH card. The FLASH card is written with the XC166 program on a PC in a card slot, using a FAT16 file system (i.e. MS-DOS). The boot program in the XC166 is able to read this format either using a home-made or commercial embedded file system.

Several modes of operation are typically used:

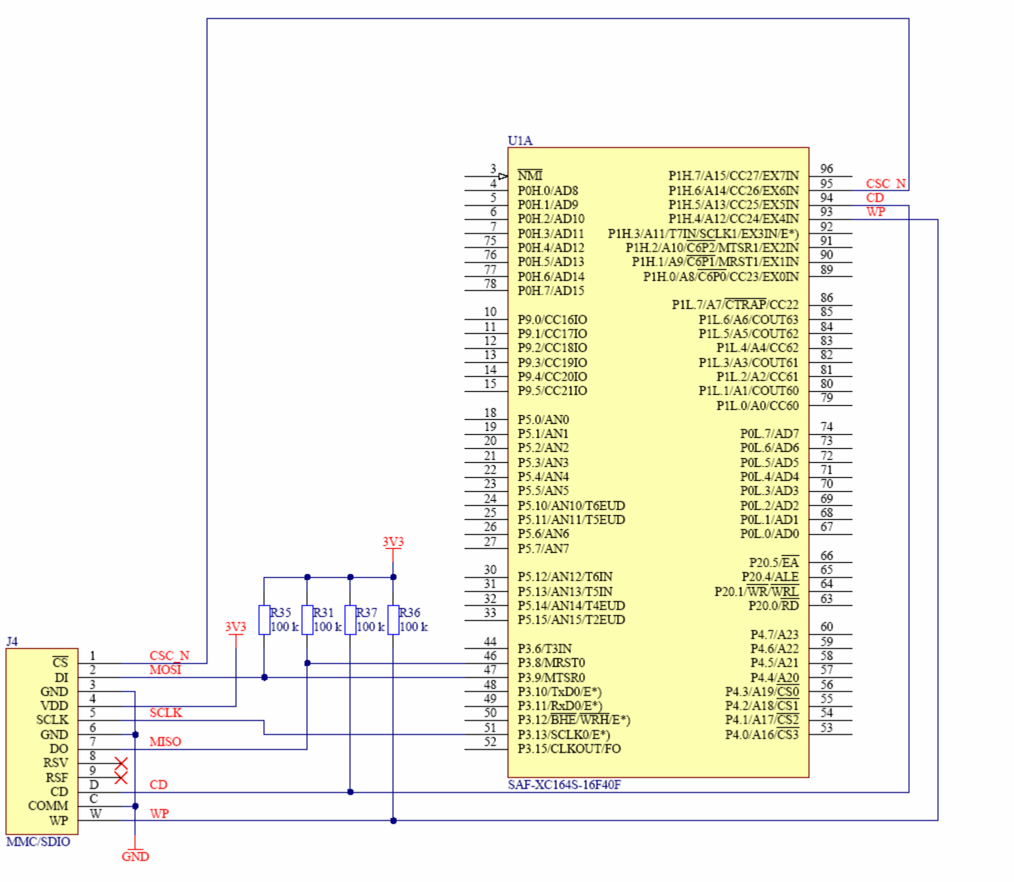
(i) The boot program loads the application from the FLASH card and writes it into a large external RAM every time the system is powered-on. This makes the XC166 appear to boot from the FLASH card, although it does not. Holding the program in a large external RAM means that a significant performance loss occurs compared to on-chip FLASH operation.

(ii) When the XC166 boot program starts, if it detects the presence of a FLASH card, it checks to see whether the program in the internal FLASH is the same as on the card. If it is not, the program in the card is loaded into the internal FLASH and the application runs. This approach means that the full performance is available.

However there are a few obstacles to be overcome before a memory card can be added to a microcontroller. Firstly, the huge storage capacity usually dictates the use of some sort of file system to keep track of objects stored in the memory. Also, as cards are likely to be read or written by a PC at some time, this invariably means a file system like FAT16 (MS-DOS/Windows 9x) or FAT32 (Windows 2k/XP) is needed. Secondly, the card has to be physically connected to the microcontroller’s bus or ports. Fortunately, this is not as difficult as might be expected.

### 4.5.3 Interfacing SD/Multimedia Cards

Secure Digital (SD) cards are physically identical to MultiMediaCards and are equipped with SPI, a function supported by the XC166 synchronous serial ports (SSCs). The “secure digital” mode uses two more pins to make a total of four data lines, giving speeds of up to 10MByte/s rather than the 100kbyte/s of SPI, but it does require a special form of UART which is not present on the XC166. Thus the cards must be run in Multimedia mode. The user must provide pins on the microcontroller for TX, RX, clock, card select and memory write protect. The illustration below shows a possible connection to the SSC/SPI peripheral on an XC166.



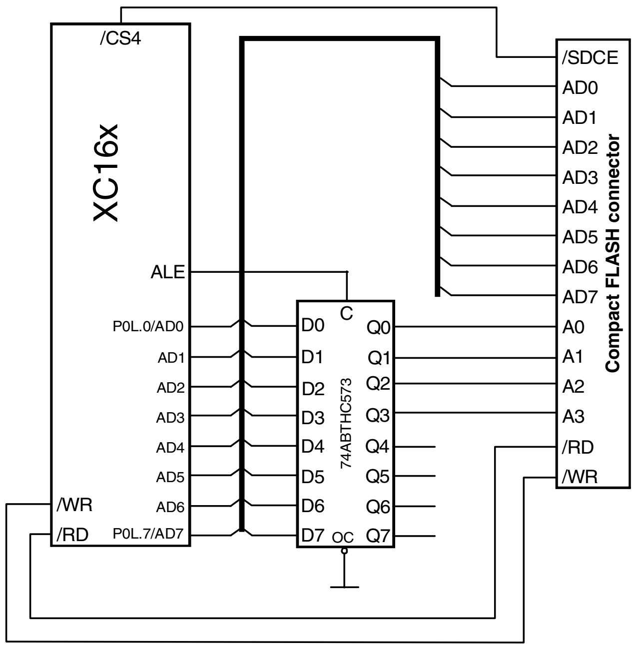

**Interfacing A SD Memory Card To The XC16x SSC Peripheral**

### 4.5.4 Interfacing To CompactFLASH

CompactFLASH cards are equipped with both the familiar IDE ("ANSI ATA") interface standard and the PCMCIA (PC card) interface found in PCs. In the latter mode, they can be addressed as simple 8-bit memory-mapped devices, which makes interfacing to a microcontroller very straightforward. The XC166 example illustrates this below. By using 8-bit multiplexed external bus mode, only Port 1 is lost, thus minimising the impact of the compact FLASH to the overall IO pin count.

Typically CompactFLASH cards require at least 5 wait-states (Phase E >= 5). They are comparatively slow devices and are used in a polled mode. It is therefore important to insert a delay between writing to a control register and then trying to read it back to check a status flag.

The software to drive FLASH cards is quite simple (you can find it at www.infineon.com/xc166-family).




**Simple Scheme For Attaching A Compact FLASH Card To An XC16x In 8-Bit Mode *(courtesy of HCC Embedded)***

### 4.5.5 Managing Large FLASH Cards

FLASH cards are not like resident FLASH in that the address of data is not held in a linear fashion. At the base of the card's address range is a bank of control registers which the user's code must use to configure the device and read and write data. The internal structure of FLASH cards is usually similar to a hard disk and indeed one common method of accessing them is via a "True IDE" mode. This is rather like a paging scheme, where an enormous memory space can be accessed through a small window using a "page number + offset" scheme. In most cases, random access to individual bytes at addresses in the card is not possible. Usually, 256 bytes must be read out into a RAM buffer in the microcontroller system and the individual bytes read or written as required. Any changes are then put back into the card by re-writing the entire 256 bytes.

All of this can make the handling of the card quite tricky and keeping track of where data is through a plethora of base pages, offsets and local RAM buffers is not easy. Therefore, in the majority of cases it is much simpler to treat the FLASH card as a file system, just like a PC or digital camera does. This allows data to be stored in named areas in a logical fashion (i.e. directories and files) and makes the retrieval of data much easier. There is also the added advantage of being able to write data onto the card in the embedded system and then being able to read it directly in the card slot of a PC and *vice-versa*.

Although simple to use, the creation of a flash file system from scratch is a complicated process. Fortunately there are a number of ready-made systems available off-the-shelf as C-source code which can handle the FLASH card hardware interface and manage the file system. Such file systems can support FAT16, FAT12 and FAT32 systems, as found in MS-DOS and Windows. Usually they will offer wear-levelling and media error handling to cope with the vagaries of real FLASH devices.

File systems are traditionally associated with large CPUs like 8086, SH4, PPC and so on, where an RTOS is often present along with huge memory resources. However this file-based approach to handling FLASH cards can be applied to single-chip 16-bit devices like the XC166.

### 4.5.6 File System API To Embedded C Programs

For many embedded programmers, having a file system present in the application is hard to visualise. If you are used to dealing with pointers and arrays, or pointers and malloc() then thinking in a file-orientated way can be very strange! Here are some examples of common file system operations in C:

Here is creating a directory on a FLASH file system in a card:

```
/* create Hello.dir */
       ret=f_mkdir("Hello.dir");
       if (ret) return _f_result(1,ret);
```

Here is making the new directory the current directory:

```
/* change into hello.dir */
       ret=f_chdir("hello.dir");
       if (ret) return _f_result(23,ret);
```

Here is opening a file for READ-only access. Here a file handle is used, just as in PC programming:

```
       file=f_open("file.bin","r");
       if (file) return _f_result(0,0);
```

Here is writing an array called “buffer” containing 512 ones into the file represented by the file handle:

```
       memset(buffer,1,512); /* set all elements in buffer to 1 */
       size=f_write(buffer,512,1,file); /* test write */
       if (size!=512) return _f_result(5,size);
```

The general format of the functions used to control the file system should be familiar to anybody who has worked with MS-DOS files in for example, Microsoft Visual C/C++.

### 4.5.7 Resources To Implement A File System On The XC166

In the XC166 CompactFLASH example given above, the EFFS-THIN (from HCC Embedded) small-footprint file system uses around 4k ROM for a minimum DOS-style file system and 20k for a complete file system, including formatting functions. Data RAM usage is under 1K with 0.5k extra for each file permitted to be opened simultaneously (“MAXFILES”). All memory is statically allocated so neither a heap or operating system is required. However many embedded RTOS like CMX-RTX and ARTX166 are able to incorporate a file system.

Providing a means to “scale” the file system according to the user’s needs and available resource is important. For example, a lot of file system code can be eliminated by omitting the FLASH card formatting function, relying on a PC to do it via a card reader instead.

#### 4.5.7.1 The Issues When Porting An Embedded File System

File systems like the EFFS-THIN are written in platform-independent C and sometimes require some configuration of the host microcontroller's peripherals by the programmer. However the XC166 is a standard configuration in the HCC range, so this is already done. For SPI-driven Multimedia cards, the user can choose to use either the XC166's own SSC/SPI interface, or the file system can provide a "bit-bashing" simulated SPI. If files are to be written with a time and date stamp, then a hardware real time clock is required.

Other typical user-defined configurations are:

- (i) random number generation and serial number generation during format - recommended only if formatting of media is required.
- (ii) semaphores for mutual exclusion - only required where a pre-emptive RTOS is present.
- (iii) which functions to exclude or include, depending on target memory resources and required features (e.g. including the formatting capability)
- (iv) selecting FAT12/16/32 support
- (v) selecting long or 8.3 name support

# 5 In-Circuit Reprogrammable FLASH EPROM

## 5.1 Introduction

The main reason for the introduction of the XC166 family was the addition of FLASH to the successful C167 series. All XC166 derivatives have on-chip FLASH in sizes from 32kbytes up to 256kbytes. Future versions will have considerably larger FLASH ROMs. The FLASH area can be programmed without a Vpp pin as 5v is all that is required. Typically the FLASH is programmed by the processor itself, even when soldered down, with the program received via the serial port's bootstrap loader mode. Please refer to section 2.5 for more information on pull-down resistors. The user's software can then receive the program as a HEX file and program it into the FLASH. It is important to note that many competitive CPUs which appear to offer the convenience of in-circuit reprogrammabilty in actual fact do not! Unlike earlier FLASH technologies used in C167 devices, the CPU can still execute code from the FLASH during programming and erase operations – should the FLASH module be busy with a program or erase, the instruction pipeline will wait for it to be ready before continuing.

It is also possible for a program to reprogram itself, i.e. receive a new version of itself and blow it into FLASH. This requires a specially written C function, which can be downloaded from the Infineon website (www.infineon.com/xc166-family).

Alternative bootstrap modes are available that use the SSC and CAN peripherals. An application note is available from Infineon that shows how FLASH is programmed via CAN. The JTAG unit can also be used for programming in conjunction with a tool like the Hitex XCFLASHer.

This self-programming ability can be very useful in cases where the FLASH CPU is to be used in mass-production, as the final program need only be put into the device at the end of the line. It also makes field software updates very straightforward. For further information on using the bootstrap loader, please refer to section 5.6.

## 5.2 Internal FLASH Layout

The XC167CI has a 256kbyte FLASH ROM, divided into four 8kbyte sectors, plus one of 32kbytes and three of 64kbytes. It is based at 0xC00000 rather than the usual 0x00000, which is also the default reset address in single-chip boot mode. It also means the interrupt vector table address (VECSEG) must be set to 0xC0. The 128kbyte versions lose the top two 64kbyte sectors.

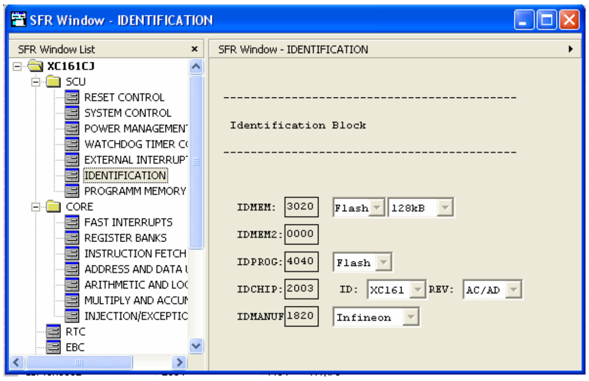
READ protection can be applied on a sector-by-sector basis or to the entire FLASH area, with a password system available to temporarily unlock protected areas. To defeat unauthorised reading of the FLASH ROM, data reads can only be made by code itself which is situated in the FLASH. Write operations can be performed in "pages" of 128bytes whilst erasing is on a sector or 256 byte "wordline" basis. As the FLASH is arranged in 64-bit rows, all XC166 instructions are FETCHed in one access, giving a significant performance increase over the classic C167's 32-bit wide FLASH. Programming typically takes 2ms per 128 byte page and a sector erase around 200ms. The entire FLASH can be erased and programmed in around 9 seconds, using a JTAG tool.

Devices are supplied from Infineon in a guaranteed erased state and unusually, this equates to a numerical zero at each location.

### 5.2.1 FLASH Identification

The IDCHIP register at address 0xF07C returns a value that allows the CPU type and FLASH silicon revision level to be determined. The values for common XC166 derivatives are listed below.

| Device Type & Stepping | IDCHIP Value |
|---|---|
| XC161-16, step AC/AD/AE: | 2003 |
| XC161-16, step BA/BB: | 2004 |
| XC164-16, step AC/AD/AE: | 2303 |
| XC164-16, step BA/BB: | 2304 |
| XC161-32, step AA: | 2001 |
| XC161-32, step BA/BB: | 2002 |
| XC164-32, step AA: | 2301 |
| XC164-32, step BA/BB: | 2302 |



**Chip Identification Information In Hitop5**

## 5.3 FLASH Reliability

Vast effort has been put into guaranteeing the integrity of FLASH data, even at the top of the 125 degree C automotive temperature range. All FLASH memory has a failure rate – defects in the tunnel oxide plus deterioration due to erasing cause charge stored in the floating gate to leak. It's just a matter of how much and when. The data retention quality required for automotive applications is at least 2 orders of magnitude greater than that needed for standard industrial or consumer applications. Automotive quality requirements are moving to ZERO defects per million.

Providing reliable FLASH for this kind of environment has proved a huge challenge to silicon manufacturers and it has taken 10 years plus for the technology to be perfected.

### 5.3.1 Dynamic Error Correction

The key to the high reliability of the FLASH is dynamic error correction. Hamming codes can detect single and double-bit errors, and correct single-bit errors as well. In contrast, the simple parity code cannot detect errors where two bits are transposed, nor can it help to correct the errors it can find.

The XC166 devices use a Hamming distance of 4 which means that the FLASH is really 72-bits wide, not 64 bits as might appear. The system allows the correction of single erroneous bits and the detection of two wrong bits. This process occurs in real time, as instructions and data are read from the FLASH. In the event of a double bit error occurring, the CPU immediately vectors to a class B trap (exception) where the user's program is expected to take some action to rectify the situation. Single bit errors do not cause a trap and are just flagged up in the FLASH Status Register (FSR) where the user may choose to carry out some action such as reprogramming the 256-byte wordline within which the error occurred. However as single bit errors are not fatal, the program proceeds as normal.

Hamming code correction of FLASH errors is required to assure automotive quality – it is needed on top of an existing high quality FLASH process and cannot be used as a substitute for poor FLASH. It must be stressed that even single-bit errors are extremely rare and likely only to occur where the FLASH has been mistreated or taken beyond its nominal 20k write-erase cycles. Double bit errors are only likely to occur at the extremes of probability, although good software engineering practice dictates that they should be handled properly.

### 5.3.2 FLASH Endurance

The stated 15 year data retention assumes 1k programming cycles (erase-programme) at the full rated operating temperature of the device. With up to 20k programming cycles, the data retention time is reduced to 5 years, worst case. With the XC166 FLASH module, these are just baseline figures as Infineon have provided a mechanism whereby the data retention can be extended almost indefinitely, albeit with a small software overhead.

An examination of typical 16-bit automotive microcontrollers reveals:

| Part | Minimum Data Retention | Write/Erase Cycles |
|---|---|---|
| XC167CS-32FF: | 15 years | 1000 |
| | 5 years | 20000 |
| DeviceX: | 20 years | 1000 |
| DeviceY: | 20 years | 100 |
| | 15 years | 10000 |

This kind of data really only gives an estimate of FLASH reliability as of course, no manufacturer could run such a test in real time. There are many factors that determine how long data will be retained, of which the number of erase cycles is usually the most significant. The rigorousness of test regime and the methodology used is also critical and this does vary from one silicon manufacturer to another. The numbers above are based on a qualification test of 1008 hours at 150 degrees C and when de–rated for say room temperature, equate to over 1000 years of data retention. However they are really just very well-informed estimates and as such, indicate that all microcontroller FLASH is very reliable these days. In practice, silicon manufacturers cannot store wafers for many months to prove sub 1 defect per million quality levels. This is where the hardware error correction helps to GUARANTEE automotive quality requirements – even if the uncorrected retention quality worsens slightly in a particular batch due to process parameter shifts, the ECC still guarantees the quality. Raw FLASH quality that cannot be guaranteed by the ECC can be detected very easily.

## 5.4 Managing FLASH Under Extreme Conditions

However some applications may get close to the worst-case limits stated and the nature of high-volume production is that even a very low probability event like a FLASH error will become likely, given enough units shipped. Normally the user has no way to check the state of programmed data and certainly has no easy way to correct any potential problems. All that the user can do is to provide some sort of fail-safe mode where a FLASH error will not cause a hazardous condition.

### 5.4.1 When To Manage FLASH

This is where the FLASH used in XC166 is unique, as it allows the user to check the state of FLASH cells and reprogram any where the retained charge in the floating-gate cell has changed unexpectedly. This requires a small software overhead and is only required in applications where:

- The product must be guaranteed to last more than 15 years in service
- Any sector of the FLASH will be erased and written more than 20000 times in 5 years or 1000 times in 15 years as part of the normal operation of the product. A sector used for storing adaptive constants that are updated every few days would be an example.

- The CPU will be operated outside the normal temperature specification[2] stated by Infineon and without their approval (i.e. oilwell applications)
- A large number of erase/programming cycles will occur at low temperatures (<= 0 degC).
- Very high integrity applications such as flight controls or medical systems with a very long service life are involved.

In other FLASH microcontrollers, the natural charge leakage in the FLASH over time will cause an eventual failure in a small percentage of units.
This will require the unit to be either scrapped or perhaps reprogrammed under original factory conditions, although this is not always a realistic solution. As in most cases the product in which the microcontroller is fitted will be out of warranty/fashion/use after 5 years, normal FLASH reliability is adequate. However the XC166 FLASH allows the user to extend the useful life of FLASH data, as is described in the following paragraphs
.

### 5.4.2 Dynamic Recovery From Double Bit Errors

The probability of a double bit error (i.e. not automatically compensated for by hardware error correction) is extremely low. Double bit errors can be avoided by performing a recovery operation after a single bit error has been detected. This requires the following steps:

1. Detect the 256 byte wordline containing the erroneous bit.
2. Reduce the CPU speed to 10MHz
3. Store the contents of the wordline temporarily in RAM or another FLASH sector (safer)
4. Erase the wordline in FLASH, provided the ambient temperature is above -20 degrees C.
5. Reprogram the erased wordline using two 128 byte write-page operations

**Note:** For the XC166-32F devices up to and including BB-step, plus the XC166-16F up to and including BA-step, the CPU speed must be set to 0.5MHz for the duration of the Erase wordline command, if the ambient temperature is below -20 degrees C.

The wordline of data copied from FLASH to the temporary RAM buffer will be valid because a single bit error during READ is automatically corrected via the hardware error correction. The erase and programming is done using the standard command sequences.

### 5.4.3 Predicting Future FLASH Failures

A further and even more rigorous type of check is possible that will spot single bit errors before they ever occur. It is really intended for use in production-line FLASH programmers, although it could be used in the application software and run on a regular basis. This test is based on examining the threshold levels (“margins”) for a bit in a cell to be considered to contain a ‘1’ or ‘0’ and is independent of the hardware error correction system. As unwanted changes in a cell’s charge results from charge-coupling from writes to neighbouring cells, sectors updated regularly with adaptive data might be at risk. The margin check involves reading each address in the sector to be maintained. The “margin” control would be set to “standard” at first and the value read from the location temporarily stored in RAM. The margin would then be set to the low level and the read repeated to see whether any bit previously read as ‘0’ has become a ‘1’. If there was a problem that causes a double bit error, a class A trap would be generated in exactly the same way as if the margin were to be normal.

This indicates a change in charge in the FLASH cell that means that this bit might continue to wander and become read as ‘1’, even using the standard margin. A single bit error would eventually occur, possibly with a double bit error as the final outcome. The read is then repeated with the margin set at the upper limit to see if any ‘1’s have become ‘0’s.

Given the very remote possibility of even a single bit error, this FLASH “routine maintenance” procedure need not be carried out very often. It might take the form of a service procedure run during the periodic or annual

[2] Infineon will permit XC16x devices to be used for 1500 hours at ambient temperatures of up to $T_A$ = 140degC, subject to the junction temperature $T_j$ not exceeding 150degC, the clock 20MHz or less and no FLASH reprogramming above $T_A$ = 125degC.

maintenance of the system incorporating the XC166 microcontroller e.g. visit to a car servicing outlet for an automotive application. Alternatively the program could perform these checks continuously as a normal background housekeeping operation.

Another use for the margin control facility is to allow wordlines that give double-bit errors to be recovered; by reducing the margin, it is possible that reads that previously gave double bit errors to be repeated to only give single-bit errors. The recovered data would then be re-programmed as in the procedure given above.

## 5.5 Tools For Programming FLASH

Infineon produce a freeware programmer for all their FLASH microcontrollers (including the 32-bit Tricore). This is MEMTOOL and is available for download from www.infineon.com. This assumes that the host PC has a serial port, which unfortunately many do not, especially laptop computers. Some USB-COM adaptors will work so some experimentation may be required.

MEMTOOL also works with K-Line (ISO1314), given a suitable RS232-K Line interface.

There are some commercial tools that allow FLASH programming via the CAN interface. For user's own solutions the Infineon FLASH-On-The-Fly application note gives an example that is reasonably easy to adapt.

### 5.5.1 Programming XC166 FLASH In Production

Whilst MEMTOOL is a very quick and useful laboratory FLASH programming tool, it is not suitable for high-volume production programming. This is especially true where the XC166 application itself is high-integrity or engineered and tested to a quality standard like IEC61508 SIL2 and above.

#### 5.5.1.1 Hitex FLASHXC.DLL Bootstrap-Loaded Production FLASH Programmer

In cases where JTAG cannot be used either because it is not tracked-out or is rendered inaccessible, the serial bootstrap mode is a good way to program the FLASH in production. The FLASHXC.DLL programmer uses a Microsoft DLL to perform the programming function when called from an user-defined front end running on a PC. This might be a programming station PC that is part of the production line control system. The DLL simply slots it and takes care of bootstrapping the processor, decoding the XC166 program's HEXfile and its transmission to the XC166 system.

The DLL is called via:

```
result = BootLoadFLASH(Hex_File_Name,
                       COMport,
                       NULL,
                       NULL,
                       KLine,
                       baud_rate,
                       secondary_baudrate,
                       PLLCON_value,
                       XTAL_freq);
```

This programmer has been independently validated and is suitable for high volume production use. Typical programming times are 40 seconds for a 256kbyte binary image.

#### 5.5.1.2 Hitex FLASHCANXC.DLL CAN-Based Production FLASH Programmer

Where the serial port bootstrap mode cannot be used, there is a CAN-based equivalent available that uses a Softing CAN-USB PC to CAN interface to communicate with the XC166. Functionally it is very similar to the serial version.

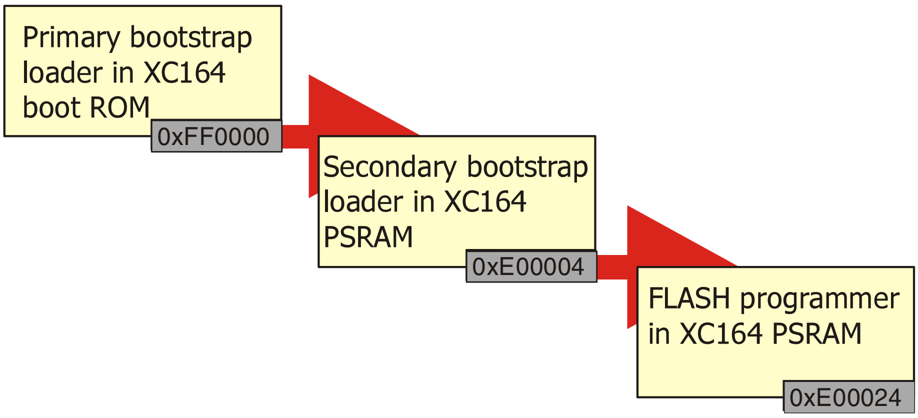
#### 5.5.1.3 Hitex XCFLASHer XC JTAG Production Programmer

For programming multiple XC166-based systems in parallel, JTAG is the best choice. This requires a special tool such as the Hitex XCFLASHer that allows up to 16 systems to be programmed simultaneously through USB. The typical programming time for a serial FLASH programmer is around 45 seconds for 256kb FLASH image whereas with JTAG, this comes down to around 7 seconds.

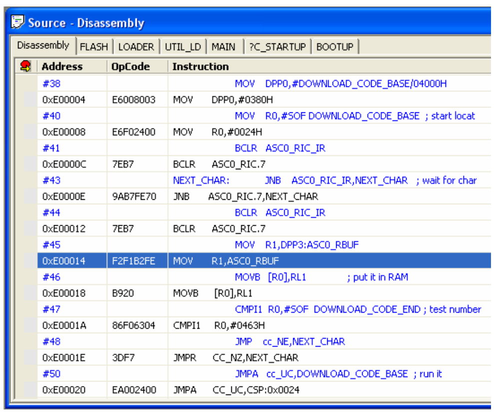
# 5.6 Introduction To Bootstrap Loader Programs

## 5.6.1 The XC166 Serial Bootstrap Loader

Like its C167 forebear, the XC166 series has a serial bootstrap loader on serial port ASC0 that is commonly used as a means of programming the on-chip FLASH both during program development and series production. This loader program is made available in a special CPU mode, entered by the CPU powering up with the /RD and ALE lines held low and the /EA high (single chip start). ASC0 is then monitored by a normally-hidden bootstrap loader program located at 0xFF0000. This sets up the processor to a minimum usable configuration that has the watchdog disabled, the serial port ready to receive and transmit and the TxD0 pin set to output. On seeing a character of one start bit, eight data bits of zero and one stop bit, the bootstrap loader measures the baud rate and then proceeds to place the next 32 bytes received into the PSRAM at 0xE00004. These 32 bytes are in fact a simple XC166 program that represents a “secondary” bootloader whose job is to receive a much larger program. Finally the “primary” bootstrap loader jumps to this address and runs the user’s 32-byte program.




In most cases, the larger program received by the secondary loader is a complete FLASH programming system that will eventually receive the final 128kbyte or 256kbyte application and burn it into the XC166 FLASH ROM. The size of the FLASH programmer is limited by the size of the PSRAM (2k in some variants) and the secondary loader is limited by Infineon’s bootstrap loader to 32 bytes, thus these programs must be tightly coded and located in memory very precisely. Also they must operate directly on the CPU’s FLASH memory controller in a manner prescribed by the silicon manufacturer with no room for misunderstandings. An additional complication is that the XC166 comes out of reset with the CPU clock divided by 2 so that typically it is only running at 4MHz. At such a low frequency, the bootstrap loader cannot auto-baudrate detect reliably above 9600 baud. Therefore the downloaded FLASH programmer must change the PLL and serial port control registers to get the CPU clock up to something more reasonable like 40MHz, allowing 115.2kbaud to be used for the program transmission. Writing such programs can be quite a test of the engineer’s competence and familiarity with both the CPU and the compiler!


Source - Disassembly

Disassembly | FLASH | LOADER | UTIL_LD | MAIN | ?C_STARTUP | BOOTUP

| Address | OpCode | Instruction |
|---|---|---|
| #38 | | MOV DPP0,#DOWNLOAD_CODE_BASE/04000H |
| 0xE00004 | E6008003 | MOV DPP0,#0380H |
| #40 | | MOV R0,#SOF DOWNLOAD_CODE_BASE ; start locat |
| 0xE00008 | E6F02400 | MOV R0,#0024H |
| #41 | | BCLR ASC0_RIC_IR |
| 0xE0000C | 7EB7 | BCLR ASC0_RIC.7 |
| #43 | | NEXT_CHAR: JNB ASC0_RIC_IR,NEXT_CHAR ; wait for char |
| 0xE0000E | 9AB7FE70 | JNB ASC0_RIC.7,NEXT_CHAR |
| #44 | | BCLR ASC0_RIC_IR |
| 0xE00012 | 7EB7 | BCLR ASC0_RIC.7 |
| #45 | | MOV R1,DPP3:ASC0_RBUF |
| 0xE00014 | F2F1B2FE | MOV R1,ASC0_RBUF |
| #46 | | MOVB [R0],RL1 ; put it in RAM |
| 0xE00018 | B920 | MOVB [R0],RL1 |
| #47 | | CMPI1 R0,#SOF DOWNLOAD_CODE_END ; test number |
| 0xE0001A | 86F06304 | CMPI1 R0,#0463H |
| #48 | | JMP cc_NE,NEXT_CHAR |
| 0xE0001E | 3DF7 | JMPR CC_NZ,NEXT_CHAR |
| #50 | | JMPA cc_UC,DOWNLOAD_CODE_BASE ; run it |
| 0xE00020 | EA002400 | JMPA CC_UC,CSP:0x0024 |

**Typical Secondary Bootstrap Loader**

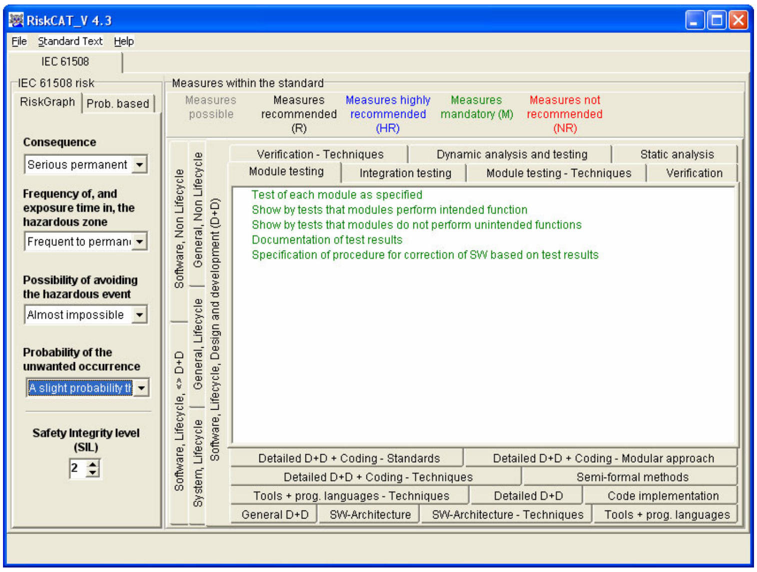
If the FLASH programmer is ultimately to be used for programming finished products on a high-volume production line, then the program will need to be extensively tested, as errors could result in complete chaos and huge costs if thousands of units are wrongly programmed. Unfortunately it is very common for quickly thrown-together FLASH programming tools to find their way from “casual” laboratory R&D use into a production environment! Programming tools provided by Infineon are not approved for production use and so most companies are forced to write their own tools from scratch.

## 5.7 Testing FLASH Programmers Under IEC61508 And Other Standards

The constraints on bootstrap-loaded FLASH programmers outlined above make their formal testing a significant challenge. A simple system-level test is relatively easy to perform as it probably consists of just “send a test application to be programmed into FLASH and see if the application runs properly”. However this type of test is not usually adequate for anything but the most trivial of applications and is certainly not sufficient for a programming tool destined for series production. In any sort of proper software testing regime, techniques such as unit testing and coverage analysis are mandatory. If the application to be programmed on the production line is itself safety-critical then the programmer needs to be tested to the same level. Thus for example, an application conducted to IEC61508 at SIL2 must include unit (i.e. module) testing and by implication some form of coverage testing.

Inevitably FLASH programmers are split into two distinct parts. Firstly there is a PC application that initialises the XC’s bootstrap mode and sends a HEXfile version of the application via a serial port to the XC166 target. Secondly there are two small XC166 programs sent by the PC end, running in the XC’s PSRAM.

The testing of the PC program is relatively straightforward as the Microsoft Developer Studio debugger can be used for simple tests and there are a multitude of commercial tools available for unit testing PC software. The real challenge is in the debugging and testing of the XC code as special tools are required if the job is to be done properly.



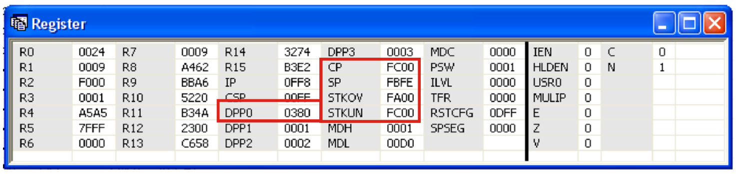

**Unit Testing Requirements For IEC61508 SIL2 And Above (RiskCAT)**

## 5.8 Bootstrap Mode Debugging Problems With JTAG

Whilst it is just about possible to debug a bootloaded-FLASH programmer by waving port pins up and down or other crude methods, it is virtually impossible to test and verify its correct operation. Using a debugger is pretty well essential but with a basic JTAG debugger like the Keil uLINK or even an advanced one like the Hitex TantinoXC, there is a fundamental problem with debugging bootstrap-loaded programs in that the JTAG is disabled by the CPU in bootstrap mode until the primary loader has completed and the user’s secondary loader has started. Thus in this mode it is not possible to get control of the CPU while the secondary loader is being received and written into the PSRAM. A partial solution to this is to write a small application that mimics the Infineon integral bootstrap loader and manually blow this into FLASH using the MEMTOOL utility. When the CPU boots up in single-chip mode, the program monitors the serial port for the zero byte, measures the baud rate, loads the 32-byte secondary loader and so on, just like the built-in loader. The risk with this method is that the user’s bootloader emulation function might not quite set the CPU up like the Infineon one and there is a high probability of incorrectly initialised CPU registers and stacks, all of which could result in unpredictable behaviour later on.

## 5.9 Bootstrap Mode Debugging Using An In-Circuit Emulator

The best solution is to use a proper XC166 in-circuit emulator like the DPROBEXC. This is based on a special form of XC device that has additional pins for external CPU control and program tracing. However unlike a traditional “bondout” chip, this “emulation device” has real FLASH memory. It also contains the real Infineon bootloader so it behaves identically to a production XC166 device of the same step level. Thus the state of the stack, CP, serial port, DPPs and other critical registers is entirely representative of standard silicon and so boot-loader-dependent programs developed on the DPROBEXC will also run first time on production hardware. This combination of features makes it possible to properly emulate a XC166 device in bootstrap mode and allow the erasure and programming of representative FLASH memory. It therefore is feasible to set a breakpoint on the exit of the user’s secondary bootloader and break as the FLASH programmer is called. Following on from this, single-stepping the actual FLASH programming lines is possible and importantly, programming errors can be simulated by intercepting the FLASH status register check that occurs after writing a wordline or erasing a sector.

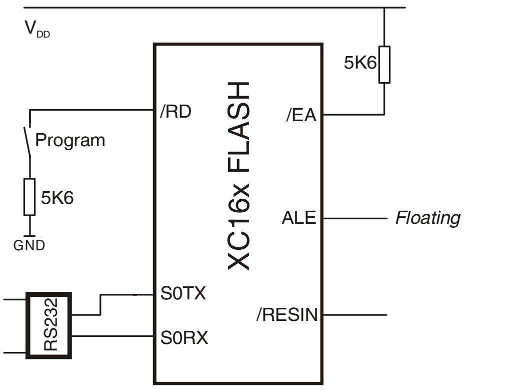

Register

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R0 | 0024 | R7 | 0009 | R14 | 3274 | DPP3 | 0003 | MDC | 0000 | IEN | 0 | C | 0 |
| R1 | 0009 | R8 | A462 | R15 | B3E2 | CP | FC00 | PSW | 0001 | HLDEN | 0 | N | 1 |
| R2 | F000 | R9 | BBA6 | IP | 0FF8 | SP | FBFE | ILVL | 0000 | USR0 | 0 | | |
| R3 | 0001 | R10 | 5220 | CSP | 00FF | STKOV | FA00 | TFR | 0000 | MULIP | 0 | | |
| R4 | A5A5 | R11 | B34A | DPP0 | 0380 | STKUN | FC00 | RSTCFG | 0DFF | E | 0 | | |
| R5 | 7FFF | R12 | 2300 | DPP1 | 0001 | MDH | 0001 | SPSEG | 0000 | Z | 0 | | |
| R6 | 0000 | R13 | C658 | DPP2 | 0002 | MDL | 00D0 | | | V | 0 | | |

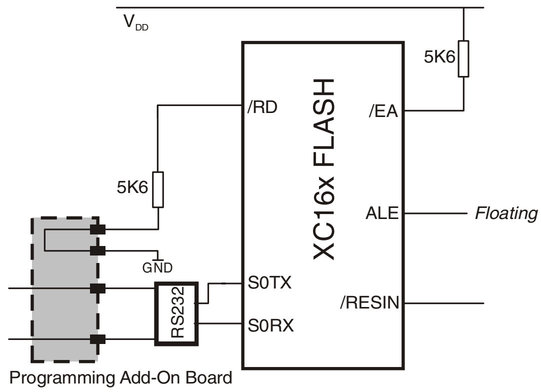
**CPU State Configured By Bootstrap Loader At Entry Of Secondary Loader**

## 5.10 Hardware Aspects Of In-Circuit FLASH Programming

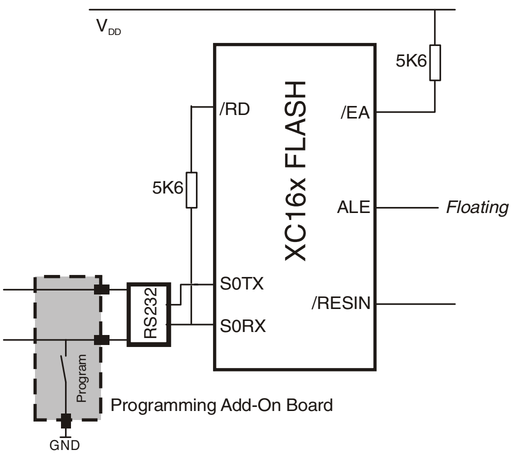
The capability of programming FLASH EPROM (external and on-chip) *after* the XC166 has been soldered down is extremely useful. To program the device, access to the serial port 0 is required plus access to the /RD and ALE pins whilst coming out of reset. The following internal start examples show various means of doing this.



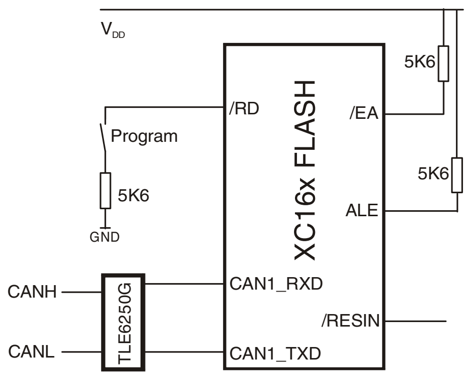

**Ex1: Simple Method Of Entering Bootstrap mode**




**Ex2: Using Special Serial Connector To Enter Bootstrap**




**Ex3: Holding The RX Line Down To Enter Bootstrap**




**Ex4: TwinCAN Bootstrap Mode**

The example **Ex1** uses an RS232 link as might be connected via a MAX232 (or similar LT7011 etc.) to a PC. The XC166's bootstrap mode is entered by a simple push button on the /RD pin. The host PC sends a null byte with one start and stop bit until the device replies with an acknowledge byte of 0xD5. The user must then send a 32-byte loader program, which then receives a FLASH programming utility which then receives the user's application program. As the integral bootstrap loader and the second 32-byte program are necessarily very simple, only binary code can be sent to them, which is not supported by any of the commercial compilers. A number of HEX to binary convertors are available to bridge the gap.

If a single-line physical communication layer such as RS485 or ISO-K is used, the S0TX and S0RX are effectively connected together. The integral bootstrap loader disables the receive side until the acknowledge byte has been completely transmitted so that it will not be confused with the first byte of the initial 32-byte program.

**Ex2** shows a means of dispensing with a manually operated bootstrap switch. The serial connector has two additional pins that short the pull-down resistor to ground so that after the next reset, the XC166 will enter bootstrap mode.

**Ex3** has the S0RX pin connected to the /RD pin via a resistor so that if the host holds its TX line down as the XC166 comes out of reset, bootstrap mode will be entered. Alternatively the serial port connector has a momentary press button switch that can ground the RS232 RX pin. This will cause the pull-down resistor on /RD to put the XC166 into bootstrap mode on the next reset. When the switch is released the serial port operation is unaffected by the presence of the resistor.

**Ex4** has the ALE pulled high as well as the /EA to enter the alternate TwinCAN bootstrap mode, whereby the XC166 expects to receive a message of ID 0x555 to enter the bootstrap loader. This mode is often just referred to as an "Alternate Bootstrap" mode in XC166 documentation. Section 5.11 gives more information the use of the TwinCAN bootstrap mode.

**Note:** The SSC0 bootstrap mode is not available in internal start mode (/EA = 1).

## 5.11 In-Circuit FLASH Programming Via CAN Bootstrap Mode

In many applications it is desirable to be able to perform the in-circuit FLASH programming function via the CAN module. The XC166 has an alternate bootstrap mode where, instead of waiting for a program to be received via the serial port, an 11-bit CAN message is expected with identifier 0x555. The CAN bootloader uses this alternating 0/1/0/1…. sequence to measure the bit times and hence the baud rate and receive an initial 5 data byte payload. This data comprises:

1. A value for the BTR low field in the CAN Node A bit timing register.
2. A value for the BTR high field in the CAN Node A bit timing register that when combined with byte 0, allows a higher baudrate to be used.
3. The low byte of an acknowledge ID that the XC166 is to send after initialization.
4. The high byte of an acknowledge ID that the XC166 is to send after initialization
5. The number of messages that will be sent to the XC166 before the bootloader will try to automatically run the downloaded program.

When the baud rate is detected, the device acknowledges the message sent by the programmer (drives an active bit in the acknowledge bit slot). The device then sends a handshake message to the programmer at the baud rate defined by the BTR values received in the initial message from the programmer. The device is then ready to receive the specified number of messages. The messages have to be pure binary code (i.e. no addresses), as the program is downloaded one message after another to PSRAM. After receiving the defined number of messages a Watchdog reset takes place and the code in PSRAM starts running. It is recommended that the first instruction downloaded into the PRAM is a DISWDT instruction to prevent watchdog timeouts resetting the device. The data bytes contained within the next CAN frames sent (number defined in the first CAN message as described above) are stored into the internal PSRAM, starting at address 0xE00000. For maximum efficiency, the data should be sent in groups of 8 bytes as per the maximum allowed per CAN frame.

In most cases, the CAN messages would originate from a PC equipped with a CAN card such a Softing CAN-AC2-PCI. A program would need to be written though to manage the sending of the XC166 application's HEXfile via CAN.

## 5.12 In-Circuit FLASH Programming Via SSC0 Bootstrap Mode

It is possible to make the XC166 boot from an external 25xxx series SPI serial EEPROM. Both 8- and 16-bit addressing is supported, however this is only true in external start mode i.e. /EA pin is low. The XC166 assumes that it is the master in master mode and that the SPI mode is 0 (0,0). It reads the chip ID and length codes from the EEPROM, using a nominal 1MHz clock and in 8-bit mode. Specifically, it reads EEPROM address 0, at which it expects to find a value of 0x5A. The position of this byte allows either the 8 or 16-bit mode to be detected. The next address (0x0001) contains a value representing the user's program length divided by 16. Finally, the program is loaded to XC166 FLASH address 0xC10000, from the EEPROM address range 0x0002 to 0x1FE2.

The user must arrange for the EEPROM chip select to be connected to XC164 Port 3.6.

A complete description of the SSC0 bootstrap mode can be found in application note AP3203411, available at www.infineon.com/xc166-family.

## 5.13 In-Situ FLASH Programming Without Bootstrap Mode

In some applications, it is not convenient to ever use bootstrap mode after manufacture. Thus whilst the program can be blown into FLASH using bootstrap mode in the controlled conditions in the factory, some other method is required in the field.

One approach is to simulate the operation of the bootstrap mode within the application. Here the application contains a function that looks for the zero bootstrap loader initialisation byte on serial port ASC0 and replies with the same "0xD5" that the integral bootloader uses. When the system is to be reprogrammed, the application simply calls this function. An example program that includes this capability can be found at: www.infineon.com/xc166-family.

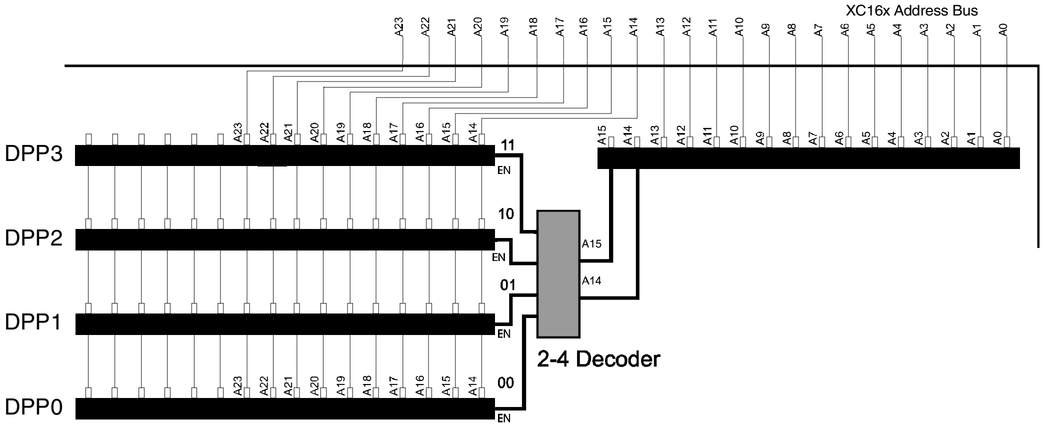
# 6 Planning The Memory Map

Planning the memory map is not difficult but it helps to know something about how the XC166 handles data accesses if you are to get the best out of it!

## 6.1 Understanding The DPPs

### 6.1.1 Fast DPP-Based Data Access

The XC166 uses the concept of 16KB long data “pages” to allow the accessing of data. Memory addresses that are within a page may be addressed by 2-byte (25ns) instructions like MOV R1,8000H. By limiting the addressing capability of individual assembler instructions to an address within a page, execution speed can be improved over other CPUs which allow 32-bit address accesses to be made in one instruction.




**Hardware View Of How The DPPs Work**

The XC166 actually only deals in 14-bit addresses that are in reality offsets from the base address of the current 16KB data page. The top two bits of any 16-bit address indicate to the CPU which DPP is to be used to form the physical address that will appear on the 166’s address bus. For example, the assembler instructions below will use DPP2 to form the physical address as the top two bits of the number ‘#8002H’ are ‘10’, i.e. 2, indicating that the page number held in DPP2 must be used:

```
MOV          R4,#8002H  ;
MOV          R1,[R4]    ;  Access address indicated by the contents of R4
```

If DPP2 contains 2, the base address of the page will be `4000H x 2 = 8000H`. Thus the address placed on the bus will be: `4000H x 2 + 0002H = 08002H`. However if DPP2 = 8, the instruction sequence would access address: `4000H x 8 + 0002H = 020002H`

Thus it can be seen that the page indicated by DPP2 can be placed anywhere in the 16MB memory space. In effect, the top two bits of the address cause an address translation. To use DPP1 for the access, the instruction sequence would look like:

```
MOV          R4,#4002H  ;
MOV          R1,[R4]    ;  Access address indicated by the contents of R4
```

Now the top two bits of ‘#4002H’ are ‘01’, indicating that DPP1 should be used. The precise mechanism that decides what the top two bits of the address are need not be of concern to the programmer, as they are calculated by the linker. Further information on using the DPPs can be found in the publication “An Introduction To Using The C Language On The 166 Family” at www.infineon.com/xc166-family.

In the case where a check sum is to be performed over a 256KB EPROM, one of the DPPs - usually DPP0 - has to be incremented every time a page boundary (0x4000) is crossed.

It must be stressed that the use of the DPPs is *totally transparent* to the C programmer under normal circumstances and need only be taken into account when absolute maximum speed is required. It should not be confused with the simple paging schemes used on some smaller HC11/12/16 type processors! As far as the user is concerned, the DPP concept should be considered as a means of creating 16KB "islands" of very fast access in the XC166 memory space. However it can be overridden for handling large address ranges, as covered in the next section.

### 6.1.2 Accessing Large Data Objects

A second data addressing mode is available that is more suited to coping with potentially very large data objects i.e. large arrays and structures. This allowed 32-bit addresses to be handled directly so that the XC166 could be regarded as having a 32-bit linear address space. Inevitably the speed of access is reduced to a small extent but it must be borne in mind that the XC166's native addressing mode is exceptionally fast! As an example, using the XC166's linear addressing mode, a 128KB block copy in external memory can be performed in 42ms at 40MHz.

The DPP mechanism was retained to permit the user to create 16KB regions of very fast access within an overall linearly-addressable memory space. The programmer therefore has the option of being able to create variables that can be addressed by the optimal method - in simplistic terms "small and very fast" or "big and fast".

### 6.1.3 Data Addressing Impact On C/C++ Compilers

The design of current XC166 C/C++ compilers is that they largely manage the various data addressing modes transparently. The only manifestations of the underlying structure are the "near" , "huge" and "xhuge" (or equivalent) keywords that can optionally be used to place variables in specific areas, determined by their size and speed of access.

## 6.2 Planning the Memory Map and Configuring the C Compiler.

Overleaf is a typical memory map for an XC166 design. Where applicable, a sample line of C is given on the left hand side to show how data can be directed to the appropriate memory region or type. The example shows an external RAM fitted. This is often added as the on-chip RAM may not be sufficient for applications with a lot of data.

The addresses of the on-chip ROM, RAM and SFRs are fixed by the XC166 silicon. The external RAM can be mapped to any location that is not occupied by one of these. Be aware that if an external memory device overlaps the address range of an internal resource, the internal data will take precedence.

### 6.2.1 Using The DPPs

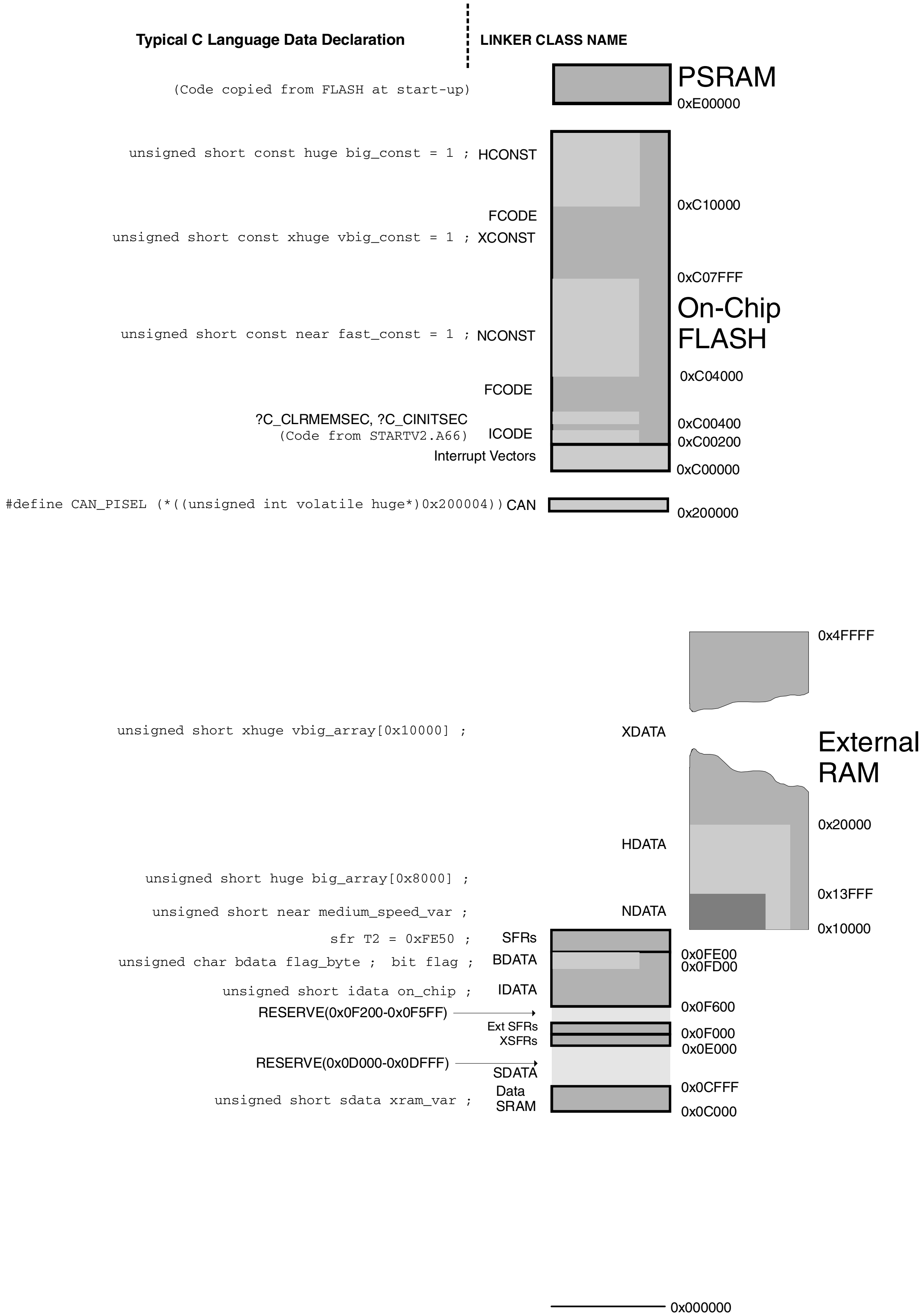
The DPPs allow four 16KB regions of fast access to be created in the memory map. DPP3 is always set to 3 so that the on-chip SRAM and peripheral control registers have the fastest access. Another DPP is set to 0x301 to create a 16KB fast region in the FLASH and a third one is set to 4 (in this example) to give a fast area in the external RAM. In this scheme, the final DPP is unallocated but it could be set to 0x302 to give a 32KB fast area in the FLASH or set to 5 to give a similar 32KB region in the external RAM. For applications where there is less than 16KB of fast variable or constant data, it could be set to 0x80 to speed up access to the CAN module.

The exact method for setting the DPP values and hence the 16KB fast region base addresses is determined by which compiler is being used.

### 6.2.2 Using the PSRAM

This area is generally only used for executing code during FLASH programming. It is designed specifically for code although it can be used for variable storage but it is slow when used this way as it takes 3-6 cycles for a read. The IDATA at 0xF600 takes 0 cycles!

To get a function into the PSRAM requires it to be stored in the FLASH but copied at startup. This means that the storage address is not the execution address. All commercial XC166 compilers support this possibility.




**Typical External RAM Memory Map**

## 6.3 External Bus Factors

The overall throughput of the XC166 family is highly dependent on the location of the code and if it is in external ROM, the bus mode used.

## 6.4 Internal FLASH

The on-chip FLASH has a 64-bit internal bus to the CPU so that all instructions can be fetched in a single access. This configuration consequently has the highest throughput of all.

## 6.5 External ROM

As a 16-bit machine, the 16-bit modes are obviously the most efficient: most instructions are 2-byte and so a complete instruction can be fetched with just one access across the bus. With the non-multiplexed mode no latching of the address is required and so the CPU can run to best effect. Branch instructions are 4-byte and so these require two accesses to fetch.

In the 8-bit mode a minimum of two bus accesses are required to fetch any instruction. Thus CPU performance is considerably reduced and this mode should only be used for accessing constant data in ROM.

With such a high clock speed the access time of memory devices is crucial. At a 40MHz clock, typical 29F800BB 70ns FLASH EPROMS require 3 waitstates (i.e. PHASE E = 3 in the XC166 bus cycle).

The change in CPU performance with bus mode, as observed on an embedded C test program, is summarised below:

| Bus Mode | Run Time (us) | Normalised | Conditions |
|---|---|---|---|
| Internal FLASH | 45.42 | 1.0 | 40MHz, 1WS |
| 16-bit de-multiplexed | 63 | 1.4 | 40MHz, 3WS |
| 16-bit multiplexed | 127.17 | 2.8 | 40MHz, 3WS |
| 8-bit multiplexed | 154.43 | 3.4 | 40MHz, 3WS |
| C167CS-LM 16-bit nonmux | 66 | 1.5 | 40MHz, 3WS |
| 80C52 | 2232 | 49.1 | 12MHz |

**Notes:** Measurements taken on a DPROBEXC in-circuit emulator. The benchmark consisted of a mix of 8, 16 and 32-bit operations plus two- and one-dimensional interpolations.

## 6.6 Implications Of Bus Mode/Trading Port Pins For IO

16-bit non-multiplexed, the fastest bus mode, is also the greediest in terms of processor pins. In this configuration both Port 0 and 1 are solely concerned with the data and address bus, in that order. This allows a very simple memory system as the address pins of the EPROM are wired to Port 1 and the data pins to Port 0. As this is not multiplexed, no 74x573 latch is required, although the ALE pin will continue to operate.

If the number of IO pins is critical, it is possible to free up the 16 pins of Port 1 by going to a multiplexed bus. The 16-bit variant is to be preferred for the reasons given above. Now Port 0 will emit the address followed by ALE going low to latch it into the 74x573. The data is then emitted to complete the access. This will slow the CPU down somewhat but by careful software design the effects can be minimised. Steps to take might be to place all frequently accessed data into the on-chip (“idata”) RAM where the multiplexing will have no effect. The 16-bit modes do, of course, require 16-bit memory devices or at least HIGH and LOW 8-bit memories. In the latter case the BHE (byte-high enable) and A0 pins can be used to select between high and low devices, as in section 4.

If cost is important, the 8-bit non-multiplexed mode allows simple (and single) 8-bit devices to be used without an address latch.  This mode is very popular - remember, the basic CPU throughput is so high on XC family devices that some performance can be sacrificed to reduce cost.

Finally, the 8-bit multiplexed mode is really only provided for accessing 8051-type peripheral devices.  Although the performance loss is around 200%, an XC166 running at 12MHz will still outperform an 8031 device by a factor of 10-15, especially if 16- and 32-bit operations are frequent.

It should be noted that if an 8-bit bus is being used, the upper half of Port 0 (“Port 0H”) can still be used as general-purpose IO.

# 7 Power Consumption

In many projects power consumption is critical, particularly in battery-powered applications. Ultimately it is the "processing power per milliamp" that is important in this situation. When comparing different processors for low-power applications this ratio can be quite difficult to arrive at and is often not taken into account. Most design engineers simply compare the maximum current consumption of competing processors and just choose the one with the lowest figure. However, as hardware engineers, they may not have considered the clock speed required to get the processor throughput needed by the application. Unfortunately this depends on other factors, such as the programmer's skill and the efficiency of the C-compiler.

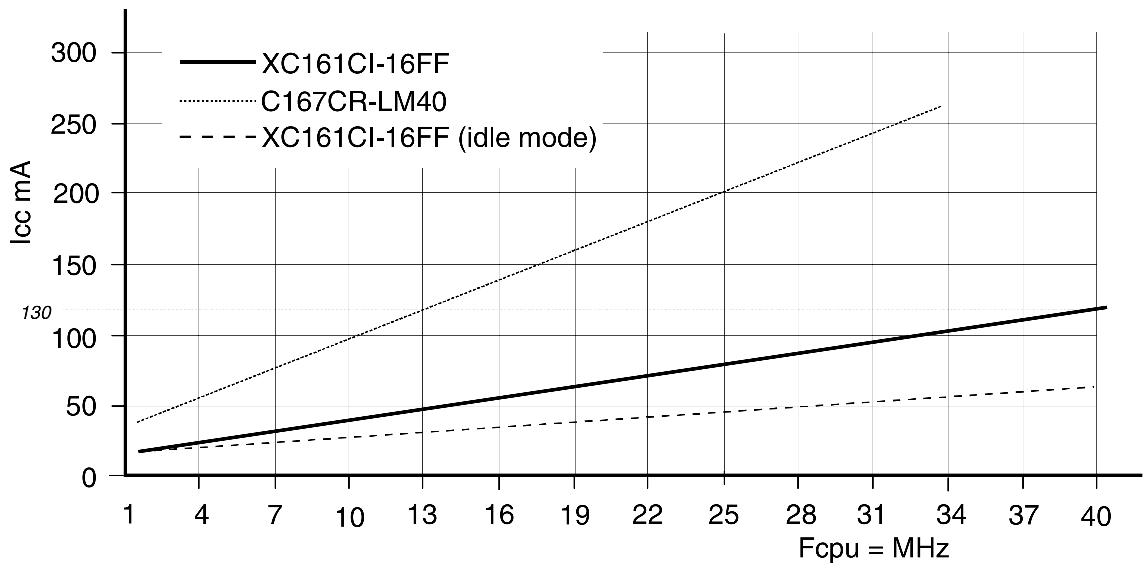
The 0.22um technology used for the XC166 fabrication and the improved design means that a mid-range XC166 derivative like the XC161CJ-16FF uses just under half the current of its C167CR-LM40 forebear, itself already an efficient CPU. A key fact to bear in mind is that the current consumption for the XC includes a 128KB FLASH device! The current for the external FLASH on the C167CR would represent another 60mA or so for a typical 29F800B FLASH.

## 7.1 Reducing Power Consumption By Optimising Clock Speed

The XC161CJ-16FF has a maximum current consumption related to the clock speed by the approximate formula:

**Icc = 10ma + 3 x Fosc**

For example, if the XC161CJ-16FF throughput is double that of another CPU, the clock can be reduced by a factor of two to get the same performance. The current will then reduce to 52% of the original figure. Thus the "C-lines per second per milliamp" of the XC161CJ is better, making it a good choice for the application. In practice, the XC161CJ is around twice as fast when programmed in C than most competitors, making it a very low-current device.




**Comparison Of The C16x/XC16x Max Current Consumption**

***(Icc = sum of IO and core current, peripherals enabled but not active)***

## 7.2 Comparing Current Consumptions

It can be very difficult to compare the current requirements of competing microcontrollers. The XC161CJ-16FF quoted here is running from internal FLASH with the peripherals enabled and active i.e. doing useful work . Code execution was from the FLASH to ensure that it was active and drawing current. Some microcontrollers are tested with the CPU executing code in a `while(1)` loop which exists only in a cache or RAM so that the FLASH is inactive. Sometimes the peripherals were active rather than just enabled. These tricks can easily remove 40mA from the total. If the microcontroller you are evaluating seems to have a suspiciously low current consumption, read the small print in the datasheet very carefully!

## 7.3 Supply Voltage

| Parameter | Symbol | min. | max. | Unit | Notes |
|---|---|---|---|---|---|
| Digital supply voltage for core | VDDI | 2.35 | 2.7 | V | Active mode, Fcpumax = 40MHz |
| Digital supply voltage for IO pads | VDDP | 4.5 | 5.5 | V | Active mode |

Note that the core supply voltage should never be more than 0,5V above that of the ports ($V_{DDP}$-$V_{DDI}$ < -0,5V). This is particularly important when switching on the power supply.

# 8 System Programming

## 8.1 Serial Port Baud Rates

The XC166 asynchronous serial ports are considerably enhanced compared with earlier C167 devices. Besides standard asynchronous (RS232-style) and synchronous (SPI-like) modes, there are also now receive and transmit FIFOs and an IrDA facility. Automatic baud rate detection is also now available.

The asynchronous baud rates may be from 50bit/s to 2.5Mbit/s with a 40MHz peripheral clock. Synchronous-mode baud rates can be between 202bit/s to 5Mbit/s. The IrDA mode operates up to 115.2kbit/s.

The inclusion of a fractional divider in the baud rate generator means that baud rate errors are typically no more than 0.02%.

One point to watch is that in the synchronous mode, the data direction of the RXD pin is not automatically set by the serial port module. It must be set manually in the DP3 register, depending on whether it is receive or transmit.

### 8.1.1 The Synchronous Serial Ports

The XC166 synchronous port supports baud rates up to 20Mbit/s. It is often used for IO expansion or communications with SPI devices like EEPROM's or real time clocks. I2C devices are best connected to the 3-channel dedicated I2C peripheral.

SSC0 is also supported by an alternate bootstrap mode whereby the XC166 can boot from a serial EEPROM.

### 8.1.2 I2C Module

The IIC Module is able to participate as Master, Slave or during Multimaster mode, as Master or Slave.
When using the I2C module, it is possible to communicate via the I2C bus without overloading the CPU. The module undertakes the following tasks:

1. (De)Serialisation of the data on the bus
2. Generation of start and stop conditions
3. Monitoring of bus lines in slave mode
4. Comparison of the device addresses in slave mode
5. Bus access arbitration in multimaster mode

The CPU clock must be at least 8MHz for an extended data rate of 400 kBit/s. It is strongly recommended that the I2C module is not accessed during data transfer. The BUM flag can be employed to ascertain whether the module is currently transferring data and an interrupt should be generated once the transfer has completed. At the same time, the ISR should read and write the I2C module.

If an interrupt is not used at the end of transmission, a short delay should be inserted after writing data on to the bus before attempting to read the IRQD flag that indicates when the transmission has finished. This is because the XC166 is able to check the flag before the I2C module has even begun transmission and the flag may be set from a previous event. This sort of problem does not occur on slow CPUs!

```
I2C_vGenerateStartCondition(); // BUM = 1: generate start condition

// Send device address  to slave
// Write device address to transmit buffer

I2C_vWriteData(I2C_ucDeviceAddress | I2C_WRITE_ACCESS);

// Wait for data to be sent

I2C_TX_RX_delay() ;

// While data transmission in progress

while((I2C_uwGetStatus() & I2C_IIC_ST_IRQD) == 0x0000)
{
  ;   // Wait to end of transmission
}
```

Typical applications for the IIC-Module are communications with:

- EEPROMs
- Real time clocks
- 7-Segment displays
- Keyboard controllers
- Audio processors

## 8.2 Adding USB Communications To The XC166 Family

A lot of XC166s have found their way into products that require a connection to a PC. Often this is in some sort of instrument or measuring system where the PC is used to either control the device or transfer data to or from it. Traditionally the connection has been simple RS232 where the PC's COM port is connected via D-type connectors to a RS232 level-shifter chip like a MAX232 that hangs off the XC166's TXD and RXD asynchronous serial port pins. With the right choice of clock crystal for the XC166, baud rates of up to 115.2k could be obtained. The software to drive the XC166's serial port is pretty simple, even if interrupts are used. At the PC end, the readfile() and writefile() functions provided by Microsoft C are quite straightforward. Thus establishing a serial communications channel between a XC166 and a PC is within most engineers' capabilities.

However the majority of new PCs, especially mobile ones, do not have a conventional RS232 COM port. This makes the whole business of talking to a XC166 system much trickier. The simplest solution is to pop down to the local PC store and get an USB to RS232 converter cable. These cost around $25 and will endow a COM-portless PC with a virtual comport that can be used to talk to the XC166, much as before. However this solution is not very elegant and is expensive, adding around $20 per unit cost and of course, means that an extra cable has to be incorporated into the product. Users who expect to see a device being recognised by name by Windows when the USB is connected will be disappointed as all that will happen is that a new COMport device will be flagged up and possibly a request for the USB-RS232 cable's installation CD ROM to be inserted. All a bit haphazard and not very professional!

Fortunately there is a better way: The active device used in many USB-RS232 converters is available as a standalone chip that can be added into existing XC166 designs without too much hardware or software effort. This is the FDT232B, produced by Future Technology Devices Ltd. Essentially it is a single-chip RS232 to USB interface that takes care of the entire USB protocol, presenting the XC166's serial port pins with 5v or 3.3v logic signals. The FDT245B has a sister device, the FDT245B, that does a similar job but whose interface to the microcontroller is an 8-bit value on the data bus. In either case, the FDT chip is added to the PCB and instantly gives an USB connection possibility, all for around $2 per unit.

The details that must be considered are covered in the next few sections.

### 8.2.1 The XC166 End Of The USB Link

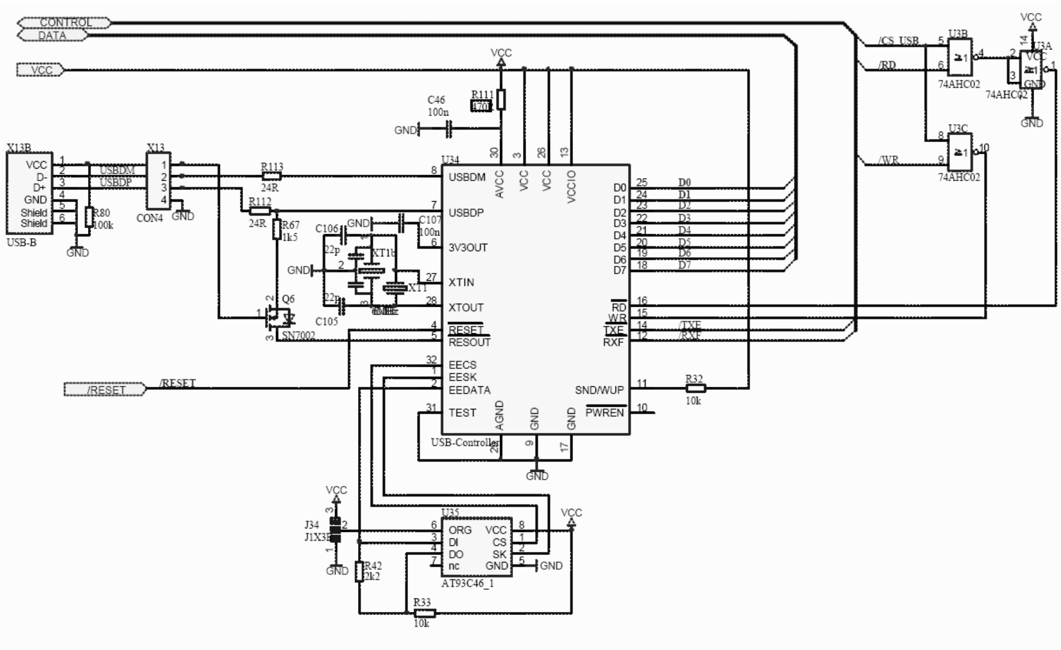
In most cases, the FDT245B is employed, even though most XC166s have two serial ports. In existing designs, the addition of USB is usually taken as an opportunity to free-up the ASC0 serial port. In new designs, the bus-mounted FT245B is preferable anyway from the point of speed of data transfer.

### 8.2.2 Hardware Issues

The FDT245B interfaces to the XC166 as shown. It is an 8-bit data device and so is connected to the D0-D7 of the XC166. The bus interface control lines are very simple. When FT245B has received some USB data from the PC, it places the #RXF pin low which tells the XC166 that data is ready to be read. There is a FIFO in the chip that can buffer up to 128 bytes of incoming data. The XC166 reads the device by putting the #RD line low. At the end of the READ, taking the line high again causes the FT245B to present the next byte in the FIFO in the same way i.e. putting the #RXE line low.

To send data to the PC via USB, the XC166 checks the FDT245B's #TXE pin and if it is low, the data can be written onto the data bus with the /WR pin low. If the FDT245B is not ready to send data over the USB, it keeps the #TXE pin high. Up to 384 bytes can be written to the device before the TX FIFO overflows. To prevent data loss, the #TXE line is held high until the FIFO empties. The #TXE and #RXE signals ideally should be routed to interrupt-ready pins on the XC166. The EX0-EX7 high speed interrupts are especially useful for this.

As there is no discrete chip-select pin on the FT245B the /CS signal from the XC166 allocated to it must be gated with the /RD and /WR signals so that the device only gets access to the bus when it is being addressed.



**Connections To A FDT245B**

### 8.2.3 XC166 Software

The software in the XC166 will simply see a stream of incoming data bytes that originate from the PC. How these bytes are used is up to the programmer. If the FT245B is being used to replace a RS232 link then the ASC0_serial TX interrupt would become the interrupt caused by the #TXE pin going low and the ASC0_serial RX interrupt would be triggered by the #RXE pin likewise. Very little code modification would be required to interface to the USB.

USB devices usually have a Vendor ID (VID) and Product ID (PID) associated with them, plus other information like serial numbers, device descriptor, class code etc.. This information is held within the device and interrogated by Windows during device initialisation so that the right software drivers can be loaded. The FDT245B relies on an optional serial EEPROM being attached to hold the application specific information like the VID and PID. Programming this is done via USB during the product manufacturing process and FDTI provide a special PC utility for doing this. The details of the enumeration and driver initialisation are outside the scope of this piece but the user is largely protected from its intricacies by the FT245B.

Although the FT245B is able to return an USB2.0 device descriptor it only runs as a full speed (12Mbit/s) device rather than a high speed (480Mbit/s) device.

### 8.2.4 PC Software

FDTI provide a royalty-free DLL-based USB interface package that interfaces directly to the Windows USB system. This is to be preferred to the simple Virtual Com Port (VCP) approach where the USB-equipped XC166 device is simply treated as an extra COM port. The package consists of a Windows WDM driver that communicates with the device via the USB stack and a DLL that is compatible with application programs written in languages such as Microsoft Visual Basic6 and C/C++ plus Borland C++ and Delphi. An INF installation file and uninstall utility are also included.

### 8.2.5 Getting Started With USB

Ready-made modules are available that allow a complete FT245B sub-system to be added to an existing design for experimental purposes for around $40. These make a full evaluation of adding USB to an XC166 relatively straightforward.

Phytec produce an advanced evaluation board for the XC166 that includes an FT245B. This makes it easy to experiment with USB interfaces and it comes with drivers for both the Windows and embedded end.

There is a huge amount of information on FDT USB interfaces at http://www.fdtichip.com

## 8.3 Interrupt Performance

The XC16 family has two methods for servicing interrupt requests. The first is a conventional, albeit very fast, vectoring to a service routine for every request. The alternative mode is to just make a single-cycle data transfer from the peripheral requesting the interrupt, with a “normal” service routine only being called once every 255 (or fewer) requests.

### 8.3.1 Interrupt Latencies

The XC166-core’s suitability for very high performance control applications derives from its combination of short instruction times and very fast interrupt response. The basic aim of the interrupt system is to get the program to the first useful instruction of the interrupt routine as quickly as possible - all stacking of current working registers (the “context switch”) is assumed to have been done before this point.

In a conventional processor, the speed of this response is normally limited by the slowest/longest instruction in the opcode set plus the time to stack the current working registers. In the case of the XC166 this might be expected to be the DIV instruction, which takes 525ns (40MHz). However this instruction is partially interruptible so that if an interrupt becomes pending during the execution of the DIV, the calculation is suspended after the first 4 cycles (100ns) so that the interrupt can be processed. Once completed, the DIV resumes for another 17 cycles. Thus the worst case interrupt latency time is not significantly influenced by the instructions in the pipeline when the interrupt is requested. The typical latency time is 425ns when running from internal FLASH with a worst case of 525ns. However for very time-critical interrupts, the special FAST interrupt mode eliminates the need to fetch the interrupt vector and so reduces these figures to 325 and 425ns respectively.

By virtue of the fast context switch scheme effectively stacking the processor state in a single cycle and vectoring to the service routine, having got to the interrupt routine, the first useful instruction can be executed immediately. Thus it takes 325ns for an XC166 to be in a position to execute the first useful line of C in an fast interrupt service routine. The availability of potentially one register bank per interrupt service routine means that each interrupt source can be considered to have its own “virtual” CPU. Provided service routine run times are kept reasonably short, this analogy is valid and can simplify the design of real time software.

To put these latencies in context, a 12MHz 8031 takes about 10us to get to the service routine plus another 12us to stack the current register set.

Note: Older CPUs with an inherently poor interrupt latency require bolt-on time-processor units. These use a micro-coded co-processor to achieve what the XC166 does using just C. Users therefore have to know not only C but also send engineers on a TPU microcode course!

### 8.3.2 Software Interrupts

In addition to the event and peripheral-driven interrupts, a large number of software interrupts are possible. These are a means of causing the program to vector to a specific routine very quickly. The priority of the service routine is the same as prevailed when the trap was triggered but the priority can be changed by the trap handler, if required via the ILVL field in the PSW register. They are extremely useful for writing real-time operating systems.

In the XC166, software traps can be assigned different priorities so that their service routines cannot be interrupted by less important events.

### 8.3.3 Hardware Traps

These are provided to allow programs to recover gracefully from major disturbances that may have caused branching to, or a word data access to, an odd address. Once in the appropriate service routine the user can decide how to deal with the upset. Of course, during program development, these traps can be a major aid to debugging - most apparent CPU oddities can be traced to the user having broken word-alignment rules or misuse of the stack!

### 8.3.4 Interrupt Structure

To allow truly event-driven software, 15 different interrupt priorities are provided, although priority zero represents the level of `main()` and thus is not considered a priority level. In fact its only use is that it will restart the CPU when in IDLE mode. Thus the response to any real time event need not be dependent on what the CPU is currently doing. Interrupts can be nested without fear of losing events and in most cases, the interrupt request flag is cleared automatically on entry to the service routine. In some CPUs different interrupt sources are grouped together so that they must have the same interrupt priority. For example, the C515's serial port interrupt is tied to the compare register 0 interrupt. Thus a real time interrupt on CC0 cannot interrupt a serial interrupt service routine, with the result that events may be lost or delayed. The user therefore cannot use an interrupt-driven serial port due to a CPU limitation.

In the XC166, no such restriction exists so that system response and performance is improved, along with an easing of program planning. As a further refinement, if two interrupts of the same priority occur simultaneously, the user can tell the CPU to which interrupt source priority can be given. The interrupt routines can change their priority level, although this is not good programming practice. Thus the XC166 interrupt structure is vastly superior to that found on similar processors and allows it to cope with a very large number of asynchronous events. It largely eliminates the need for a real-time operating system and a true event-driven system can be created using just C.

### 8.3.5 Interrupt System Usage Notes

#### 8.3.5.1 Interrupt System Pitfalls

It is not allowed to have two interrupt sources with the same priority and group level. Doing this will result in the CPU vectoring to random interrupts. If you are using a debugger, setting breakpoints on the unexpected interrupts is futile as the CPU behaves completely randomly!

Also, when the fast interrupts are used, if two interrupts use the same priority and group level the first one to trigger will work apparently correctly. However if the second one triggers, the service routine for the first one will run again. This can be very mystifying!

A detailed examination of XC166 interrupt system characteristics can be found in the Infineon application note, AP1608310 at www.infineon.com/xc166-family.

### 8.3.5.2 Critical Regions And Interrupt System Pipeline Effects

Unlike the classic C167, C166S V2 pipeline enhancements mean that clearing the global interrupt enable flag (PSW.IEN = 0) will immediately disable all interrupts. This removes the need to insert NOP instructions after clearing the flag. However this instant disabling does not apply to individual interrupt sources i.e. the Interrupt Enable flags for the various peripherals.

Where it is critical that a particular interrupt is disabled before any other operation can begin (i.e. there is a critical region), special steps are required that *do not* use the ATOMIC sequence instruction. Therefore to clear, for example, the GPT12E_T3IC_IE flag or GPT12E_T3IC_IR flag, the following C macro is recommended by Infineon:

```
#define Disable_One_Interrupt(IE_bit)   \\
{if(IEN) {IEN=0; IE_bit=0; while (IE_bit); IEN=1;} else {IE_bit=0; while IE_bit);}}
```

Usage Example:

```
Disable_One_Interrupt(GPT12E_T3IC_IE) ; // T3 interrupt enable flag
```

**Notes:**

(i) It must be emphasised that this is only required where it is critical that the interrupt cannot occur one more time after disabling it. It most situations, this should not be an issue.

(ii) If you are using a commercial real time operating system, make sure that it takes account of this CPU characteristic.

(iii) Full details of this aspect of the interrupt system are give in the latest XC166 documentation updates, available at www.infineon.com/xc166-family.

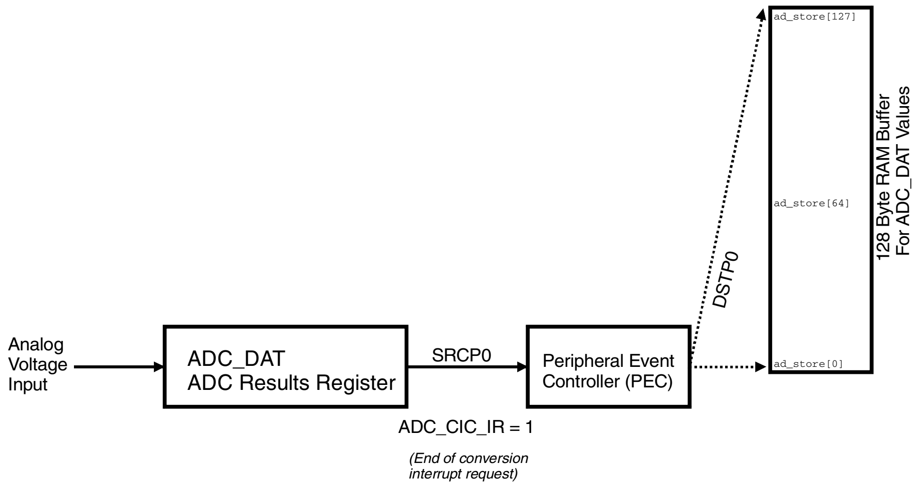
## 8.4 Event-Driven Data Transfers Via The PEC System

In addition to the normal event-interrupt routine mechanism, the XC166 also supports a special data transfer-only interrupt response mechanism. This allows a single byte or word data transfer between any two locations in single 64k segment in just a single CPU cycle (100ns). It is very similar to conventional DMA except that it is event-driven by interrupt sources. Typical applications would be to transfer A/D converter readings from the A/D results register to a buffer, without CPU intervention. This PEC system thus allows simple, repetitive data transfers to be performed with virtually no CPU overhead.

The source and destination addresses are determined by source and destination pointers in the on-chip RAM and must be initialised by the user in C. The source pointer might be the A/D result register and the destination pointer an array in RAM. If the source and destination pointers are fixed, there is no need to ever call a normal interrupt service routine and the data transfer will continue *ad-infinitum*. The source and destination for PEC-transferred data can be located anywhere but any array used must not cross a 64k segment boundary.

### 8.4.1.1 PEC Usage Examples

(i) If the A/D converter result is to be transferred into a RAM array for later processing, the destination pointer must be incremented. Up to 508 transfers can be made before a conventional interrupt service is required; however all that has to be done at this point is to reset the PEC transfer counters to 254 and set the destination pointer to the original values.




(ii) A table of values representing a sine wave are held in table. At the end of every carrier period, the associated interrupt request causes the value in the table indicated by the PEC's source pointer to be transferred to the duty ratio register of PWM module channel 0. The source pointer is then incremented by the PEC hardware to point to the next table entry. Every 128 carrier periods, the interrupt request is accompanied by a normal interrupt service routine, which sets the table pointer back to the start of the table. As a result of the foregoing, a sine-modulated pulse train appears on a port pin, with virtually zero CPU intervention. By careful software design, it is possible to completely automate the performance of complex tasks, so that the CPU is freed up for more demanding processing.

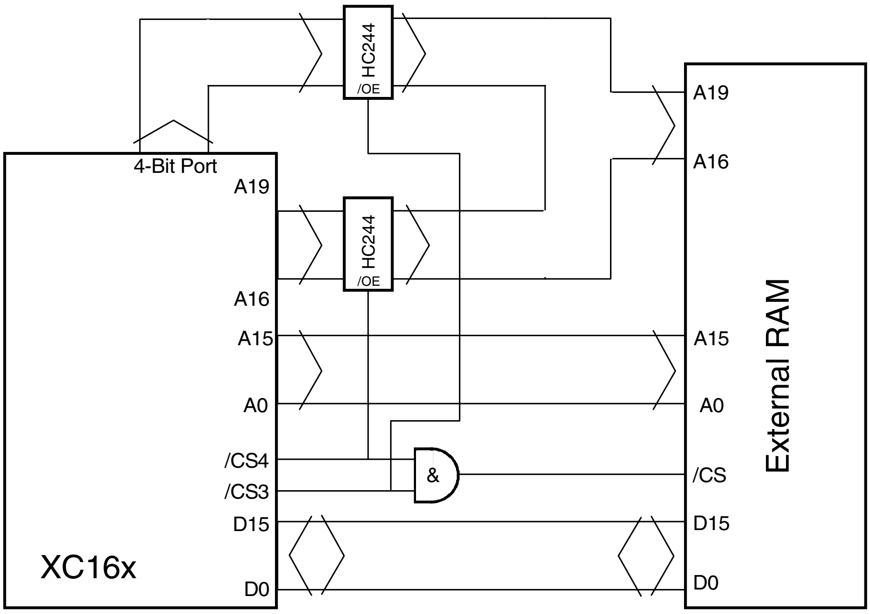
(iii) The synchronous serial port is configured to send 8 bit frames at 10Mbit/s to a second XC167 device in the same system to allow fast data exchange. Each device has a 256 byte area of RAM that is automatically copied between devices at 10Mbit/s, without CPU intervention to simulate dual-port RAM. Thus each CPU can write into this area with a guarantee that it will appear in the corresponding location in the other device within 400us. The SSC transmit and receive interrupts each use a PEC channel to send sequential bytes which repeat every 256 events.

### 8.4.2 Extending The PEC Address Ranges And Sizes Above 64K

In some applications, is it desirable to transfer very large arrays of data to a peripheral via the PEC. Such a situation can occur in inkjet printing systems whereby a 256kb bitmap image has to be moved byte-by-byte to a print-head attached to the 10Mbit/s synchronous port. The software generates the bit map image of an ASCII text string in a two-dimensional 256kb array through a low-priority routine. The first “row” in the array is filled and then transferred to the synchronous port with the PEC. During transmission, the second row of the array is filled and then it is transferred. Thus the processes of generating the bitmap and transmitting it occur in parallel, with just a single 25ns cycle stolen by the PEC on each transfer.

The basic scheme creates two images of the RAM that will be PECed to the synchronous port - one that appears as a single 512kb area at 0x80000 and the other as an 64K “window” at 0x10000 but still into the same area. The large region is created by mapping chip select /CS4 to 0x80000, length 512K, via ADDRSEL4. When the image generation software addresses the RAM, data is moved though the top HC244 bidirectional bus transceiver that is also enabled by /CS4. For the PEC transfer, /CS3 is mapped to 0x10000, length 64K. The 8-bit port effectively sets the offset of the 64K window into the RAM. The PEC’s source pointer is set to 0x0000 and the corresponding PECSEG register to 0x0801 also so that READs from the RAM cause the lower HC244 to pass the data.

The net result of this is that the job of driving the print-head is entirely automatic!




**Extending The PEC Range Beyond 64k Blocks**

## 8.5 XC166 Family Stacks

Unlike earlier C167 processors, the XC166 can support a system stack of up to 64kb, anywhere in the 16MB address space. This stack is used for the return addresses of function calls plus any PUSH or POP instructions and as there are no stack-relative data instructions, it is relatively lightly loaded. As a consequence, XC166 C compilers do not usually support stacks of more than 512 16-bit words. Programmers used to older CPUs like the 80C186 might imagine that running out of stack is very likely. In fact this is never the case, due to the multiple register bank RISC architecture and the provision of a potentially huge number of “user stacks”, based on the MOV reg,[Rx+] and MOV reg,[Rx-] instructions. These are effectively the missing stack-relative data instructions. In traditional CPUs, function parameters are pushed onto the stack by the caller, where they are either moved off the stack into local RAM or operated on directly in situ. Also, the return address must be stacked.

This has two side-effects:

(i) A considerable amount of time is spent moving data on and off the stack.
(ii) A large amount of stack RAM is required.

With the XC166, only the return address[3] and Program Status Word (PSW) are pushed onto the system stack with any parameters being moved onto the user stack, created via general purpose register R0. In practice with the Keil and Tasking compilers, the caller will leave parameters where they were, i.e. in the general purpose registers. The combined effect of both these actions is to drastically reduce the size of system stack required plus considerably reduce the processing overhead for function-calling in C. Executables of up to 2MB are commonplace on external memory designs but with system and user stack sizes of 256 and 1kbytes respectively.

In the case of interrupts, the traditional approach of stacking the current register set is possible but is not the best way. The multiple fast register bank concept allows the entire 16-register context to be switched in one 25ns cycle and with no stack use at all, other than for the return address and the PSW. On exit from the service routine the previous context (register bank) is restored. Please see the relevant section in the XC166 C Language Introductory Guide for more details on handling the stacks.

[3] In segmented mode (compiler memory models that support programs > 64k), the Code Segment Pointer is also stacked.

# 8.6 The XC166 DSP Co-Processor

## 8.6.1 Adding DSPs To Microcontrollers

It is very common to find 16-bit microcontrollers being used in conjunction with digital signal processors (DSP). Here, the microcontroller reads data from inputs and exports calculations such as FFT, IIR, FIR and other functions. The results are read back and used to drive communications devices or user interfaces.
In many situations, the DSP is lightly loaded and it is often the case that if the microcontroller had additional instructions, it could probably do the calculations itself. The main bottleneck in DSP-type operations is the long sequences of multiply-and-add (or "multiply and accumulate") operations required, particularly in filter and FFT calculations. A sample from a simple Fourier transform routine is shown below:

To reduce cost and complexity, the XC166 includes a MAC coprocessor. This allows the XC166 core to export complex calculations to a second CPU. To accommodate this, a number of new instructions of the form "CoXXX" have been added to the XC166 instruction set. The MAC allows the execution of a 32-bit operations in a single machine cycle. To simplify multiply and accumulate operations, it has its own 40-bit accumulator which allows

```
coefficients.a[0] += data_table[data_index] ;
coefficients.a[1] += (long) data_table[data_index] * ref_cos_table[sine_index] ;
coefficients.b[1] += (long) data_table[data_index] * ref_sine_table[sine_index] ;
coefficients.a[2] += (long) data_table[data_index] * ref_cos_table[2*sine_index] ;
coefficients.b[2] += (long) data_table[data_index] * ref_sine_table[2*sine_index] ;
```

sums to be accumulated over a very large number of multiply-and-add operations without overflow.
The following commands can be carried out with the MAC unit:

- 16-bit x 16-bit signed and unsigned multiplication
- Concatenation of 2 words to a 32 Bit value (read into the accumulator)
- Normalisation/scaling
- 40-bit add and subtract
- 40-bit signed addition (accumulator)
- Data limit/saturation
- Accumulator shift
- Loop count

The commands in the MAC unit are executed in parallel to the regular instructions, i.e. they are not carried out via the pipeline. The MAC opcode is executed immediately after the issuing of a CoXXX command.
It should be noted that the MAC unit makes use of the standard XC166 multiplication and division-units. Also, the MAC unit's status words must be saved separately if used in an interrupt service routine! This is usually not a major limitation as it is best to concentrate all the DSP operations in a single task or interrupt service routine anyway.

The MAC unit can be used to achieve considerably higher performance in comparison to that obtained via normal calculation in the CPU. However to get the best from it, special measures are required.

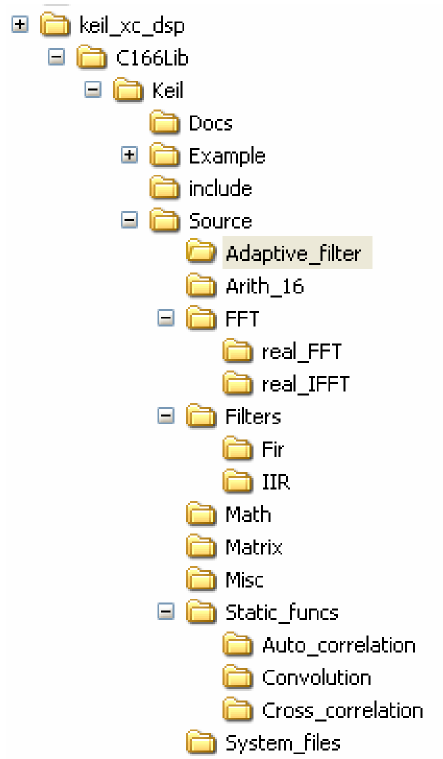
## 8.6.2 Compiler Handling Of The MAC

The C166 compilers will use the MAC to a limited extent, provided its use has been enabled. It will be used for multiplication of long (32-bit) quantities but the run time savings are very small, being in the order of 7-10%. However to get the best advantage from the MAC then the special Infineon DSP libraries need to be used.

## 8.6.3 Infineon DSP Libraries

DSP libraries are available from the Infineon website and are compatible with the Keil and Tasking compilers. They provide hand-coded assembler versions of common DSP operations such as square root, sine, IIR and FIR filters, convolution, matrix operations and statistical functions. These are typically 20 times faster than the equivalent functions coded in ISO-C and make the XC166 into a reasonable low-end integer DSP. In many cases, they will eliminate the need for external DSP devices altogether.

The libraries for both compilers are available for download at www.infineon.com/xc166-family.




**Contents Of The Infineon DSP Library**

Typical run times for common DSP functions using the libraries are:

| Signal Processing Task | Start | End | Run Time | |
|---|---|---|---|---|
| Real forward FFT, decimation in time | 12296 | 15468 | 31.72 | us |
| 16-bit finite impulse response filter | 1735 | 2506 | 7.71 | us |
| 32-bit finite impulse response filter | 1935 | 3025 | 10.9 | us |
| 16-bit symmetrical finite impulse response filter | 1755 | 2330 | 5.75 | us |
| 3x3, 16-bit signed matrix transposition | 1737 | 1890 | 1.53 | us |
| 3x3, 16-bit signed matrix multiplication | 1972 | 2477 | 5.05 | us |
| 16 bit real Convolution | 1870 | 2460 | 5.9 | us |
| 16-bit infinite impulse response filter | 1513 | 1920 | 4.07 | us |

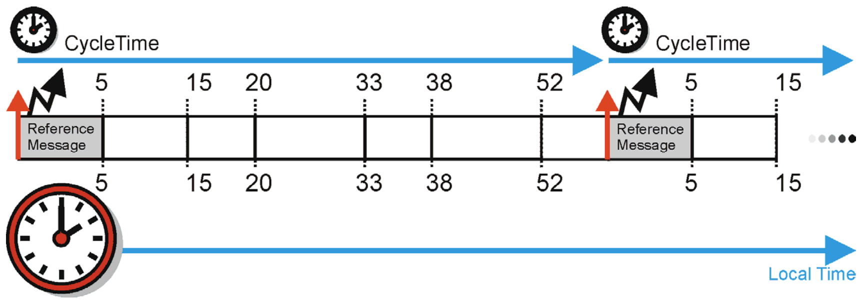
**Test Conditions:** 40MHz XC161CJ, internal FLASH, SMALL model, Keil C166 compiler, DPROBEXC emulator

The libraries are supplied as source code and are simple to integrate into XC166 C programs.

# 8.7 Special CAN Module Possibilities

## 8.7.1 Time Triggered CAN Using The TwinCAN Module

The TwinCAN module replaces the original Intel/Bosch derived module found on the classic C167. One of its new functions is the ability to support TTCAN (Time-Triggered CAN). This has not been highlighted by Infineon but the CAN controller provides TTCAN Level 1 with most of the significant features implemented, i.e. timestamp generation and single-try (i.e. no automatic retransmission).




**Time-Triggered CAN Overview**

In order for the XC166 to function correctly in a TTCAN network, it must have the following characteristics:

- There must be a timer/counter present that is incremented on each bit time.
- Received data must be provided with a time stamp. A special reference message must be recognised and stored at the beginning of the frame as a reference mark. This is used for the determination of an offset.
- Messages must be able to be transferred within a defined time window with respect to the reference mark.
- In order to avoid errors from inadvertently affecting the whole bus flow during transmission, a single try function must be implemented. This must prevent the automatic repetition of a message should an error arise or arbitration be lost.

Most of these functions have already been implemented with TwinCAN interface. However, a time window for the sending of data cannot be automatically generated. If this feature is required then it must be implemented via software. Generally, one of the biggest problems is the creation of timestamps and the sending of messages with a defined offset. Ideally this is implemented with a timer, which is started with the arrival of the message. Alternatively, one of the CAPCOM units can be used to accomplish this. If a CAPCOM channel is available then it can be utilised for the internal generation of the interrupt.

A synchronisation of network nodes in global time is allowed for in the TTCAN-Protocol (Level 2). This is implemented using the XC166 's RTC.

## 8.7.2 CAN Loop-Back Mode

A special loop-back mode is available to allow CAN node A to interact with CAN node B without any external pins being used. Thus it is possible to develop applications which only use one CAN interface by generating test messages with the other.

# 9 Allocating Pins/Port Pins In Your Application

The number of IO pins available on a member of the XC166 family varies from 47 on the XC164CM to 103 on the XC167CI. Every pin has at least one alternative function and it should be noted that these functions are not always consistent throughout the family. For example the chip select signals can be found on port 6 of the XC161 and XC167, but on port 4 of the XC164. It is often the case that at the point when the user has the least knowledge of the capabilities of the peripherals, the most critical choices regarding pin allocation have to be made, i.e. at the beginning of the project! Frequently pin assignments have to be changed in light of experience gained. The following examines what sort of functions each peripheral and then port pin is suited to. This should allow you to make the right choice for the particular signals in your application.

## 9.1 General Points About Parallel IO Ports

Each port (except P5 which is input only) can be configured to be input or output, with any combination of inputs or outputs on the same port. After reset all the port pins are high-impedance inputs and it is up to the user to program the appropriate DPx registers to turn individual pins into outputs.

When configured as outputs, ports P2, 3, 4, 7 and P9 can be either constructed from conventional push-pull drivers or open drain via the ODPx registers. The former can drive a pin either high or low, whereas the open drain type can only pull a pin low against an external pull-up resistor. This method allows an easy wired-AND and can save on external logic. The default mode is push-pull outputs. In addition, the edge characteristics can be configured by the POCONx registers, both the strength and the sharpness of the edges can be controlled (although only in groups of 4 bits).

Ports P2, P3, P4, P6, P7, P9 and P20 can in addition be programmed as inputs with custom input characteristics. The default is TTL-like input thresholds but the PICON registers allow CMOS inputs with hysteresis to be chosen. This can be useful in noisy environments or where the input level changes very slowly. Where the pin has an alternate function, the default state of the pin is a high impedance input. The alternate function is only connected to the pin as a result of the user setting up the peripheral. If the peripheral is intended to, for example, drive a square wave onto a pin, it is always the user's responsibility to set that pin to be an output, using the appropriate DPx register. For example, P3.10 is the serial port 0 TX pin. It only assumes this function if the user has correctly configured the S0CON UART control register and then set DP3.10 and ALTSEL0P3.10 to 1 to connect the UART output to the P3.10 pin.

## 9.2 Allocating Port Pins To Your Application

With a little ingenuity, it is possible to use XC166 family peripherals to generate or measure any sort of digital signal. Most peripheral blocks are able to perform even quite complex tasks without any CPU intervention - the Peripheral Event Controller (PEC) is a great help in this area. The following survey of the available port pins can only suggest some basic peripheral configurations and functions. If you are trying to use a particular peripheral to solve a problem in your application, please feel free to email us with a description of what you want to do and we'll try to come up with something!

## 9.3 Port 0

If the design requires an external bus, this port forms the data bus in non-multiplexed configurations or the combined address/data bus in multiplexed systems. In single chip applications, this is a general purpose bi-directional IO port.

If the processor is booting from external memory, port 0 hosts user-defined patterns of pull-down resistors to determine the characteristics of the bus, number of chip selects, PLL clock multiplier etc.

On the XC164CM, which is intended only for single chip applications, P0 is not present.

### 9.3.1 Port 0 Pin Allocations:

P0.0 - D0/AD0
P0.15 - D15/AD15

## 9.4 Port 1

In applications using the non-multiplexed external bus modes, this port forms the address bus. In a single chip application, this is a general purpose bi-directional IO port. On some derivatives P1 can be used for CAPCOM22 to CAPCOM27 IO. Also, P1 can be used for the Synchronous Serial Channel 1.

Both Port 0 and Port 1 contain high and low registers so that each byte of a port (for example P1L) can be written (e.g. via a PEC) without affecting the other half of the port.

## 9.5 Port 2

This port is not present on the XC164xx.

Besides being general purpose IO pins, the alternate function of port 2 is 8 of the capture and compare (CAPCOM) IO pins. The XC consisting of two capture compare units, each comprising 16-bit timers and 16 data registers. It is a means of either generating precisely-timed pulses or measuring times between events. It is analogous to the “time-processor units” found on some older CISC processors, except that it is integrated into the CPU core, rather than being bolted on as a separate processor. As the C166S V2 core is very fast and able to react to real-time events very quickly, the entire CPU is effectively available to process data connected with the CAPCOM unit. This is in marked contrast to TPU-equipped processors where only a simple microcode-driven core is available.

### 9.5.1 The CAPCOM Units

A CAPCOM unit can either “capture” the value of one of the timers or be made to toggle a pin when the contents of a particular register matches (“compares”) that of the chosen timer. Each CAPCOM register has one pin allocated to it. The channels are assigned to external pins as follows (those shown in *italics* are the only CAPCOM channels available on the XC164 due to the limited number of IO pins)

| | | |
|---|---|---|
| CC0IO – CC7IO | assigned to | P6.0 – P6.7 |
| CC8IO – CC15IO | assigned to | P2.8 – P2.15 |
| *CC16IO – CC21IO* | *assigned to* | *P9.0 – P9.5* |
| *CC22IO* | *assigned to* | *P1L.7* |
| *CC23IO* | *assigned to* | *P1H.0* |
| *CC24IO – CC27IO* | *assigned to* | *P1H.4 – P1H.7* |
| CC28IO – CC31IO | assigned to | P7.4 – P7.7 |

The capture function allows the time at which an external port pin level transition occurred to be recorded, referenced to a 16-bit timer. The edge-sensitivity can be +ve, -ve or both. Through this a wide variety of pulse measurement tasks can be realised. The compare function allows a pin to be toggled or put into a defined state when a timer reaches a defined value. The XC’s CAPCOM unit has two modes, Staggered and Non-Staggered. In Non-Staggered Mode the output signals are switched at the same time so that the basic unit of time is 25ns (at 40MHz), whereas in Staggered Mode, the outputs are switched in consecutively in 8 steps, giving a basic unit of 200ns. This mode is compatible with the classic C167 CAPCOM operation. Non-Staggered mode allows the generation of higher frequencies when in Pulse-Width Modulation mode but the fact that multiple outputs will change state simultaneously rather than with a 25ns separation can cause problems driving large inductive loads on several channels.

The input of the timers is a 25ns – 3.2us (@ 40MHz) clock (0.2us – 25.6us in Staggered Mode), derived from the main CPU clock. Timer T0 can additionally be clocked by an external signal of the user’s own choosing, applied to the T0IN pin. This gives rise to some interesting possibilities in applications such as engine management or

motor drives, when pin transitions must be created at precisely defined *angular* positions of a shaft or rotor. Effectively, when compare registers are assigned to T0, being clocked by edges from an armature position sensor, the commutation of a DC motor function can be carried out automatically.

The creation of a software UART is very simple: the capture function for a particular pin can be made to both detect the falling edge of the start bit and then clock the bit stream into a variable.

The simplest use of the compare capabilities is the generation of pulse-width modulation (PWM). When running in "edge-aligned" mode, 8-bits resolution at a 19.5kHz carrier can be produced, while an 8-bit "centre-aligned" PWM is possible at 9.75kHz, all at a clock of 40MHz. In non-staggered mode, these rise to 156.250kHz and 78.125kHz respectively. Applying modulation, usually sinewave, is very straightforward and a complete demonstration CAPCOM three-phase sinewave synthesiser driver is available from www.infineon.com/xc166-family. By adding an external low-pass filter the CAPCOM PWM channels can also be made into very accurate digital to analog convertors.

The power and flexibility of the CAPCOM unit is considerable and it is unusual to find a signal measurement or generation task that it cannot be used for!

The port also hosts 8 very high speed interrupt inputs on P2.8-P2.15, which are sampled by the CPU every 25ns. P2.15 is also the optional count input for timer T7.

### 9.5.2 Time-Processor Unit Versus CAPCOM

A point that is often missed when comparing microcontrollers is that even with the overhead of servicing the CAPCOM unit, the overall throughput of the XC166 is still large. If it is conservatively assumed that the XC166 CPU is three times the performance of a CISC processor (in reality 3-5 times is usual, depending on the benchmark used) and the CAPCOM service requires 25% of the XC166's capacity, the remaining processing power is still more than double that of the CISC. In a "properly" designed XC166 system the CPU load due to the CAPCOM is rarely more than 10%.

### 9.5.3 32-bit Period Measurements

While the CAPCOM is essentially a 16-bit peripheral, it is possible to make a 32-bit period measurement that would ordinarily require 32-bit timers and capture registers. This is achieved by using both the CAPCOM's timers, running half a period out-of-phase to generate the upper 16-bits of the 32-bit value. An application note is available from Infineon at [www.infineon.com/xc166-family](www.infineon.com/xc166-family) to illustrate the techniques involved.

### 9.5.4 Generating PWM With The XC166 CAPCOM Unit

(i) Asymmetric PWM (edge aligned)

The PWM pin goes on when the compare register matches the Timer 0 value and goes off when the timer overflows. The PWM period is determined by the value in the T0REL register. A T0REL value of 0xFF00 (-255) yields an 8-bit PWM of period 256 x 200ns in staggered mode and 256 x 25ns in non-staggered mode. The PWM on edge only moves when the PWM changes, resulting in an increase of harmonics in motor/transformer windings during duty ratio changes.

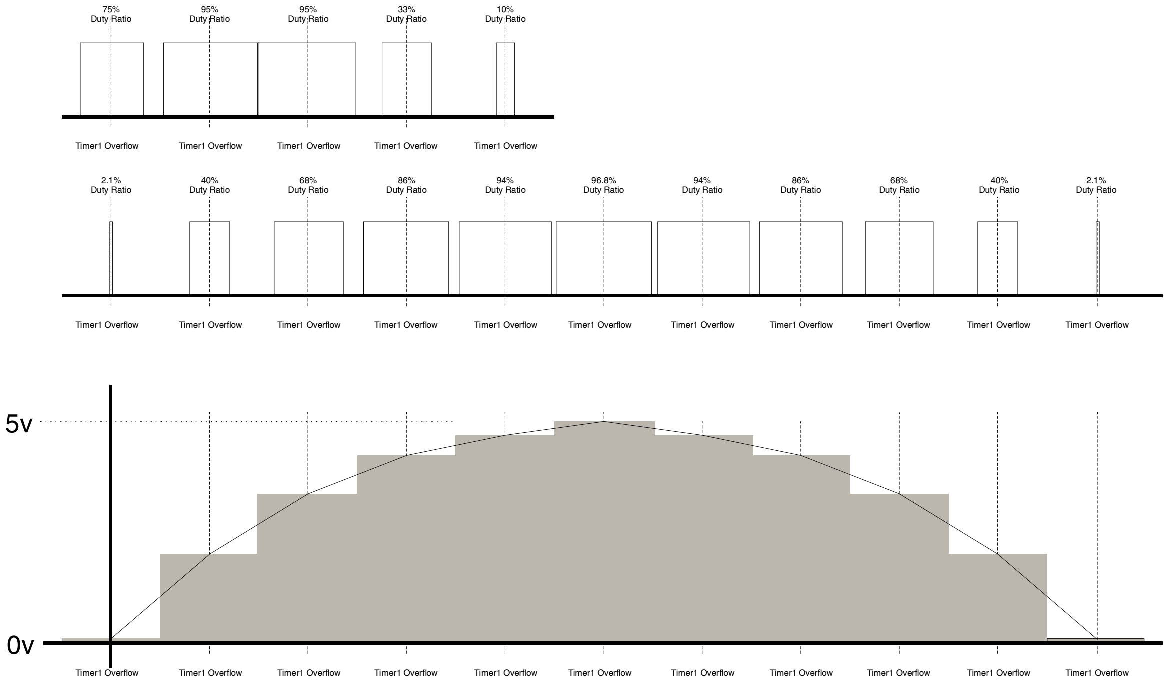
(ii) Symmetrical PWM (centre aligned)

The PWM pin goes on when the compare register matches the Timer 0 value. By using the double register compare mode the PWM pin can be made to go low again when the timer is equidistant from the reload start count. This yields a PWM waveform in which both the on and off edges move together. Thus a symmetrical PWM is created. This PWM format is to be preferred for driving inductive loads.

The PWM period is defined by the value in the T0REL register. A T0REL value of 0xFF00 (-255) yields an 8-bit PWM of period 256 x 200ns in staggered mode and 256 x 25ns in non-staggered mode.

## 9.5.5 Sinewave Synthesis Using The CAPCOM

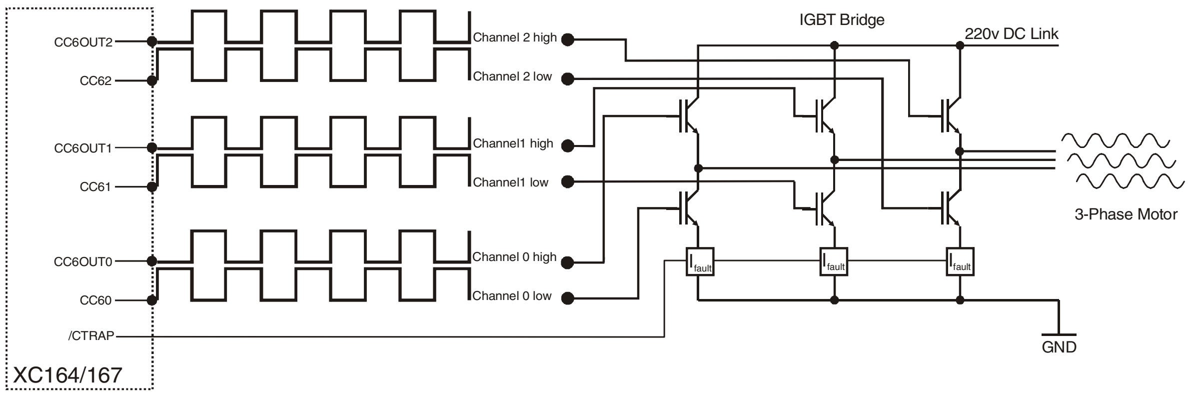
It is fairly easy to configure CAPCOM1 to produce the 6 output signals required to drive a three-phase AC induction motor. This is covered in detail in the application note, “The 166 Microcontroller As A Three Phase Induction Motor Controller”, available on request. The XC167 can drive two motors simultaneously by using CAPCOM2 as well.




**Sine Wave Generation With The CAPCOM**

## 9.5.6 The CAPCOM6E Motor Drive Peripheral

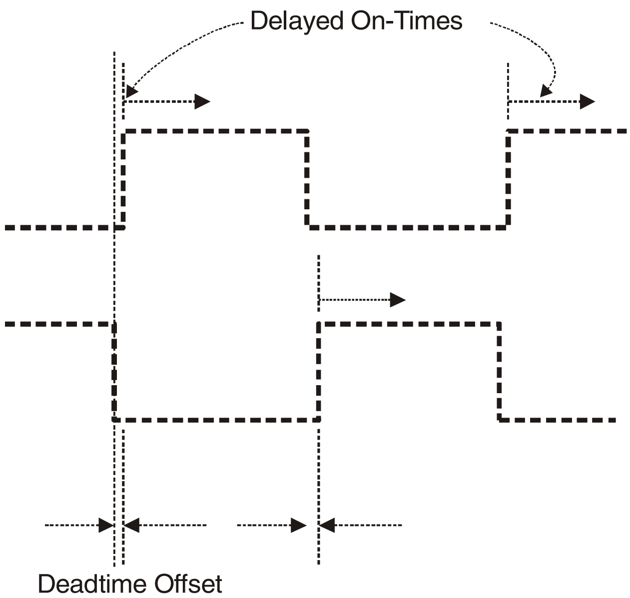
The XC164 version is intended specifically for motor drive applications, having three special CAPCOM channels with *six* outputs which make the implementation of high performance controllers easier. These are present in the CAPCOM6E peripheral, which is also included in the XC167CI. Typical motor types supported are; DC-brushless (with Hall-effect or back-EMF position sensors), AC induction (open-loop and vector), switched-reluctor plus up to 4 independent permanent magnet DC motors.




**Connecting CAPCOM6E To Half-Bridges For 3-Phase Induction Motors (Simplified)**

### 9.5.6.1 CAPCOM6E Operation

The key CAPCOM6E feature is that each PWM channel has a slave second pin which provides the complementary output for driving power bridges. The required deadtime offset for the slave channel is automatically added in hardware rather than via software, as in the standard CAPCOM units. For safety/over current trips or emergency stop, the special /CTRAP pin forces all outputs into a predefined passive state immediately, essential for the protection of IGBTs and MOSFETs configured in a bridge formation.




**Automatic Deadtime Offset Generation With CAPCOM6E**

The input to the CAPCOM6E unit is the CPU clock (or some multiple of it) for 3-phase and stepper motors but can optionally be accompanied by a three Hall effect position sensors that can be used in several ways to give block commutation in DC brushless applications. In the simple case, for a given pattern on the CC6POSx inputs, an user-defined output pattern can be placed on the output pins to advance the armature to the next position. Alternatively the PWM duty ratio can be altered progressively to produce the trapezoidal waveform required by most DC brushless motors.

The two CAPCOM6E timers, Timers 12 and 13, are usually used as the PWM carrier frequency generator and modulation timebase respectively. Timer 12 can generate compares with three compare registers that have direct connections to the CC6OUT0/1/2 pins. Timer 13 has a single but dedicated compare register and one output pin (CC63), sufficient to allow it to trigger an interrupt that hosts the code required to modulate the raw PWM produced by Timer 12.

```
void timer12_13_init_XC164(void)
{

      /* Timer 12 - base carrier generator */
      T12OF = con.deadtime0 ; // Set deadtime between low and high
                              // side PWM outputs

      T12P = pwm_period ;     // Set period register to define carrier
                              // frequency

      CTCON = 0x0000 ;        // Timer 12 runs at 0.05us per count
      CTM = 1 ;               // Up/down mode for centre aligned PWM

      /*** Configure Timer13 For Modulation ***/

      CTCON |= 0x0300 ;       // Timer 13 runs at 0.100us per count

      T13P = pwm_period ;     // Set update rate of modulation

      Set_Priority_13(T13IC) ;

      T13IR = 0     ;        // Clear any spurious interrupts
      T13IE = 1     ;        // Enable timer 13 overflow interrupt
```

**Setting Timers 12 & 13 For PWM Carrier & Modulation**

Timers 12 and 13 can run at up to 40MHz, allowing carrier frequencies of up to 156.25kHz for an 8-bit PWM resolution.

### 9.5.6.2 CAPCOM6E Applications Information

Where the XC166 application does not involve motor or inverter control, CAPCOM6E can also be used for general timebase generation, digital to analog conversion and input frequency measurement, much like the standard CAPCOM units.

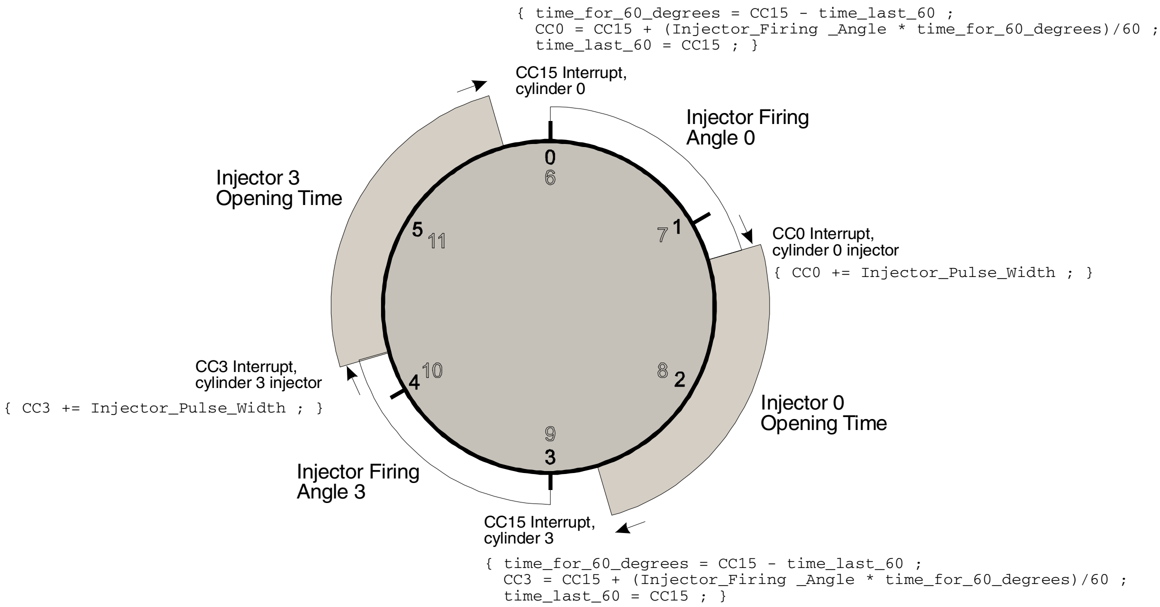
For more information, there are a number of application notes available at www.infineon.com/xc166-family that cover how the CAPCOM6E can be used to produce cost-effective, high performance motor drives.

```
void timer13_int(void) interrupt 0x4e using TIMER13_REGS {

   int CCx_temp ;

   #define CC8_image CCx_image   /* define temporary registers */
   #define CC9_image CCx_image
   #define CC10_image CCx_image
   #define CC12_image CCy_image
   #define CC13_image CCy_image
   #define CC14_image CCy_image

   /* Calculate Values For Next Interrupt */

   CCx_temp = ((long)((int)VF_con.VF_scaling) *
              ref_sine_table[current_carrier][freq0.acc_bytes[1]])/VF_Div ;

   CC60 = CCx_temp/2 + table_offset ; // Remove LSB and add table offset

   CCx_temp = ((long)((int)VF_con.VF_scaling) *
              ref_sine_table[current_carrier][freq1.acc_bytes[1]])/VF_Div ;

   CC61 = CCx_temp/2 + table_offset ; // Remove LSB and add table offset

   CCx_temp = ((long)((int)VF_con.VF_scaling) *
               ref_sine_table[current_carrier][freq2.acc_bytes[1]])/VF_Div ;

   CC62 = CCx_temp/2 + table_offset ; // Remove LSB and add table offset

   STE12 = 1 ;      // Enable loading of compare registers from
                    // shadow latch for next cycle
   CT13P = 0 ;

   STE13 = 1 ;      // Enable loading of compare registers from shadow
                    // latch for next cycle
```

**Applying Sinewave Modulation On Three Channels**

## 9.5.7 Automotive Applications Of CAPCOM1

The one-shot compare mode (mode 1) is useful for generating precisely timed pulses such as are required for fuel injection and ignition control. By driving timer 0 with edges originating from a crankshaft sensor, the compare

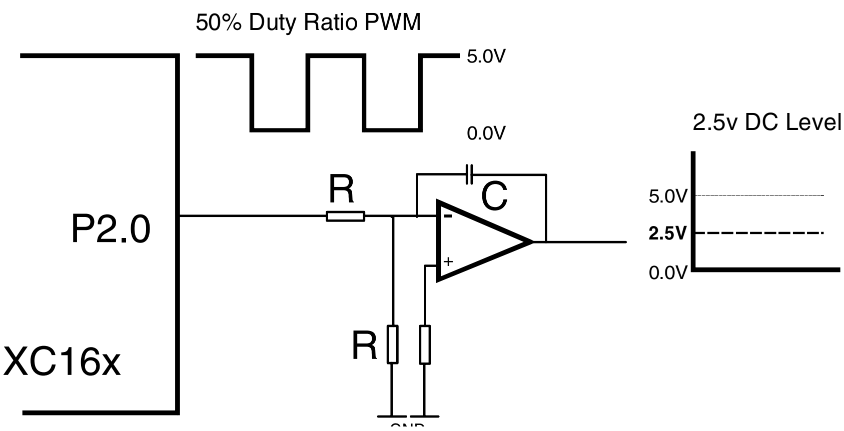



**Scheme For 12 Cylinder Sequential Fuel Injection**

registers become a means of generating pin transitions at user-defined crankshaft angles. Two application notes are available at www.infineon.com/xc166-family, that describe firstly how the CAPCOM can be used to produce 12 sequential fuel injection drives and secondly how it can be used to drive diesel unit injectors.

## 9.5.8 Digital To Analog Conversion Using The CAPCOM Unit

In PWM mode and with a suitable low-pass filter, the CAPCOM can be used to produce a very accurate A/D conversion. Where the load is inductive the load itself will average the voltage level automatically and no filter is required. In most cases a simple low pass filter, with a cut-off frequency well below the PWM switching frequency, will remove any noise and give a smooth DC level.




As might be expected, the resolution of the PWM-based A/D converter is related to the switching frequency. At 156.25kHz this would be 8-bits, while for 14-bits 2.4414kHz results. In the latter case the cut-off frequency of the filter would need to be around 500Hz. The CAPCOM6E PWM module can be used in a similar way to add four more channels.

### 9.5.9 Multiple Independent Timebase Generation

Besides being able to either drive port pins or measure incoming pulsetrains, the CAPCOM unit can simply generate interrupts at defined times. Either a conventional interrupt service routine can be called or, more often, a PEC data transfer made. It is possible to generate one timebase per CAPCOM channel so this could mean up to 32 independent events. To do this, one of the CAPCOM timers is set to free running timer mode (e.g Timer 0). The times at which the first interrupts are to occur are then placed into the 16 compare registers (CCx). When the compares occur, the first act of the service routine is to add a number to the compare register that corresponds to the number of timer counts until the next interrupt. Thus the timing of each interrupt is completely independent of the free running CAPCOM timer and the other 15 CAPCOM channels in the unit.

### 9.5.10 Software UARTs

It is very easy to add extra UARTs to the XC166 family using the CAPCOM unit. A typical single UART will represent approximately a 1% CPU load at 19200 baud, under worst-case conditions. Special software UARTs like a receiver for IRDA reduced duty-ratio infra-red links are very simple to implement. Here is a simple conventional NRZ receive routine in ‘C’ on port 2.8:

```
/*** UARTA Receive Interrupt ***/
// Detects falling edge of start bit. Waits one and a half bit periods until centre of
// bit 0, then waits one period. Sample input pin every bit period and count bits shifted
// in. After 8-bits revert to input capture mode for next start bit
/***/

void uartA_rx_interrupt(void) interrupt 0x18
{

   if(!start_bit_detect_mode)
   { // If jump not taken, time is saved???

    /*** Now in centre of bit period so sample uartA pin ***/

      rxA_shift_reg_input = uartA_input_pin ;
      rxA_shift_reg = rxA_shift_reg >> 1 ;

      CC8 += SABRG ;   // Make interrupt one bit period later

      uartA_bit_count– ;
      if(Z) {
        uartA_bit_count = 9 ;
        start_bit_detect_mode = 1 ;  // Enable CC8, capture neg edge, T0, to
                                      // find next start bit
        SARBUF = rxA_shift_reg ;
        SARIR = 1 ;      // Set dummy receive interrupt pending flag
         }
      }
   else
   {

      /*** Start bit detected... ***/

      CC8 += SABRG + SABRG/2 ;  // Wait 1+1/2 bits until
                                // first input pin sampling point for bit 0
      start_bit_detect_mode = 0 ;  // Enable CC8, compare mode 0, T0
      }
   }
```

## 9.6 Port 3

In addition to providing general purpose I/O, this port is connected to the GPT1 and GPT2 timer blocks. These 5 timers can be combined in various ways to implement gated timers, counters, input capture, output compare, PWM and pulsetrain generation, plus complex software timing functions. The XC164 has no Port 2 CAPCOM unit pins and so these general purpose timers are of special significance. They allow the XC164 to generate and detect real-time events, despite their more microprocessor-like appearance.

P3.0 - CAPCOM Timer0 count input / Serial port 1 receive
P3.1 - Timer 6 toggle latch output
P3.2 - Capture of Timer5 input/reload of Timer 6 input
P3.3 - Timer3 output
P3.4 - Timer3 count direction control
P3.5 - Timer4 count/gating/reload/capture input
P3.6 - Timer3 count/gating input / *Serial port1 receive on XC164/EEPROM chip select for SSC0 bootstrap mode*
P3.7 - Timer2 count/gating/reload/capture input
P3.8 - 165/7/1 Synchronous serial port master receive/Slave transmit
P3.9 - 165/7/1 Synchronous serial port master transmit/Slave receive
P3.10 - Serial port0 transmit
P3.11 - Serial port0 receive
*P3.12 - Bus high enable or /WRH – Not on XC164CM*
P3.13 - Synchronous serial port clock
P3.15 - System clock output

Another very important point regarding P3 on the XC164 is that P3.1 – P3.5 and P3.7 are also used for the On Chip Debug System (OCDS discussed in more detail in chapter 12) so this can be a restriction when using this member of the family.

### 9.6.1 Using GPT1

GPT1 consists of three 16-bit timers (T2, T3 & T4) plus a number of I/O pins. It can be used to make period measurements, generate pulsetrains or PWM. Like the CAPCOM unit, it is based on a maximum input frequency of 10MHz ($F_{CPU}$/4). The T2IN, T3IN and T4IN input pins can be used as clock sources for their respective timers. T2IN and T4IN are also able to trigger a capture of the free running timer 3. If the timing functions are not required, the GPT1 input pins, T2IN, T4IN and T3IN, can be used to generate interrupts. There are dozens of different ways of using GPT1 but what follows are typical applications. Ones marked with an '*' are available as application notes.

*Some typical GPT1 applications are:*

- PWM driver for DC motor drives*
- X-Y trackerball input position detector*
- Timebase generator*
- 33-bit period measurement
- Missing tooth detector*
- Automatic baudrate detector up to 115.2kBaud*
- Quadrature encoder input - provides speed and direction of quadrature input with zero CPU overhead*

### 9.6.2 Using GPT2

This is a block of two 16-bit timers that can operate at up to 20MHz on a 40MHz CPU. The input clock can be a pre-scaled version of the CPU clock or an external input signal, up to 5MHz. Like GPT2, the timers can be concatenated to produce a 32-bit timer. However it can do some very clever tricks, such as the multiplication of an input frequency applied to the CAPIN pin or period measurement with zero CPU intervention.

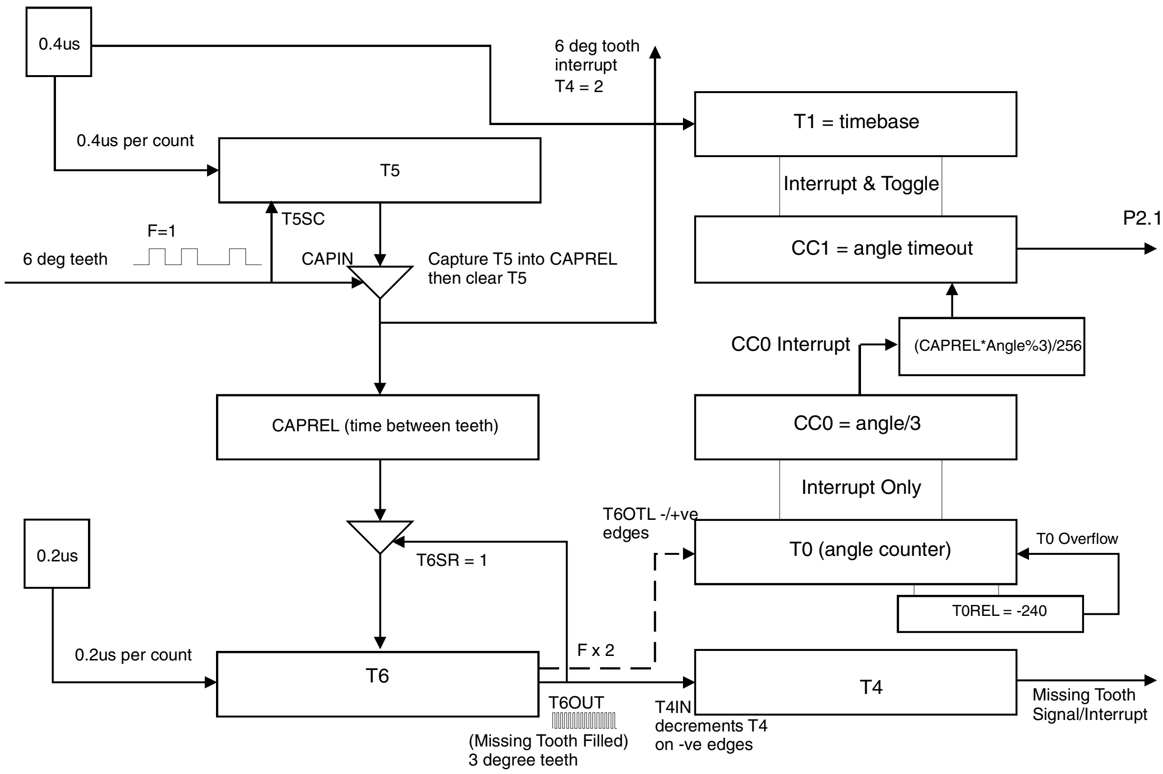
*Some typical GPT2 applications are:*

- Timebase generation*
- Pulse generation
- Time-between-edge measurements*
- Two-channel software UART*
- TV line capture and buffering*
- Automotive missing tooth filler*
- Pulse position modulation receiver for TV remote control*

### 9.6.3 Combining CAPCOM, GPT1 & GPT2 For Crank Synchronisation

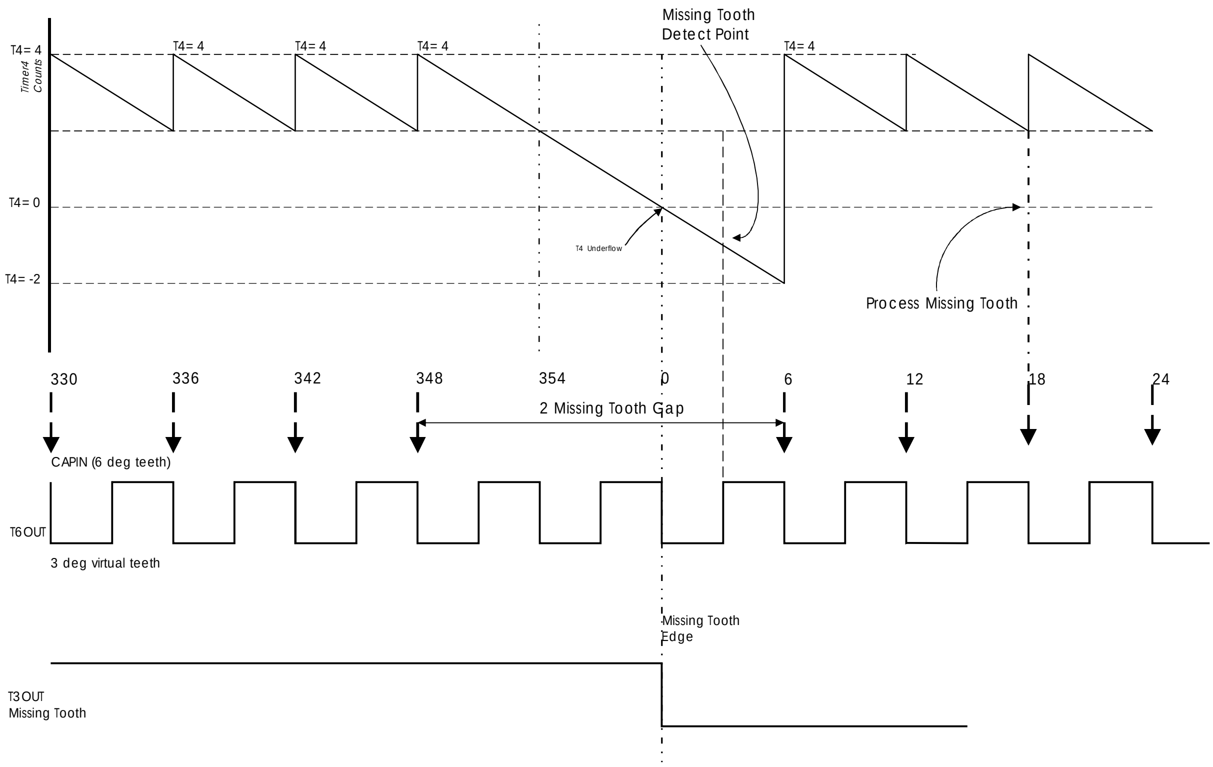
It is possible use the CAPCOM unit, GPT1 and GPT2 together to perform the basic crankshaft synchronisation and angle-domain injector (or ignition) firing process with a very low CPU overhead. The key to this is an apparently redundant connection between the Timer 6 output (T6OUT) and the CAPCOM Timer 0 input.

In the scheme shown below, the crank shaft input is connected to the CAPIN pin and GPT2 used to automatically generate a value in the CAPREL register that represents the time between the crankshaft teeth and doubles the tooth rate. In this case, the 6 degree teeth are transformed into “virtual” 3 degree teeth. This higher rate tooth stream is then used to drive CAPCOM Timer0 so that it becomes a crank angle counter. Setting the T0REL (Timer0 reload) register to –240 means that Timer0 will automatically count the 720 degrees of a 4-cylinder, 4-stoke engine without any user intervention.




**Scheme For Crank Synchronisation And Event Generation With Almost Zero Software Intervention**

The missing tooth is detected by using GPT1. This works by reloading Timer 3 with a constant (here 16) on every real 6 degree crankshaft tooth but with T3 being clocked by the 3 degree tooth stream. The direction of the Timer 3 count is controlled by the (T3EUD) pin so that it count up when the input is high and down when it is low. When a missing tooth occurs, the reloading of Timer 3 does not occur but the 3 degree teeth do as GPT2 acts as a PLL. This will cause Timer3 to underflow and thus cause an interrupt. The underflow also causes the T3OUT pin to toggle, thus providing an edge on every missing tooth. This process will continue *ad infinitum*, with no software overhead.




**Missing Tooth Detection With GPT1**

The injector opening and closing events are scheduled by using compare registers (CCx) to detect when a certain angle has been reached. This angle can be resolved to the nearest 3 degrees. The value of the angle compare register will be `CCx = (required angle)/3`. This accuracy is not sufficient for either injector or ignition control so any remaining angle needs to be converted into the time domain. This is done by changing the compare register mode to compare against Timer1, which is configured as a simple free-running timer. The angle to time domain conversion is simple and requires the calculation `CCx = (CAPREL % (required angle))/256`.

This system will work, provided that the engine speed does not more than double in a single tooth period, something which is very unlikely!

## 9.7 Port 4

This port is not present on the XC164CM.

Port 4 is a general purpose digital I/O port whose alternate function is that of the segment address lines in applications that require an external bus. In addition, on the XC164CS and XC164N/D/S variants, the chip select signals are also present on this port. The number of segment address lines and chip selects used by an application can be set up either by the Port 0 settings if the processor boots from the external bus, or by software in single chip applications. The CAN signals can also be routed through Port 4, although there are options to also use Port 9 and 7 (XC167 only).

### 9.7.1 Interfacing To CAN Networks

#### 9.7.1.1 Allocating Pins To The TwinCAN Module

The CAN peripheral's TX and RX alternate functions of P4.5 and P4.6 (as well as both P4.4 and P4.7 if the second CAN port is used) may appear to limit the addressing range of the XC166 to A0-A19, i.e. 1MB. This is not the case as in fact up to 5MB can be addressed by using the up to 5 available chipselects. The CAN RXD and TXD pins for both halves of the TwinCAN module can also be routed to alternative ports, as shown below. Any combination of RXD and TXD pins may be selected and has can be seen, Port 4 pins do not need to be used at all so no loss of address lines need occur.

| TwinCAN Signal | Default Pin Set | IO Pin Set 2 | IO Pin Set 3 | IO Pin Set 4 |
|---|---|---|---|---|
| RXDCA | P4.5 | P4.7 | P7.6 | P9.2 |
| TXDCA | P4.6 | X | P7.7 | P9.3 |
| RXDCB | P4.4 | P9.0 | P7.4 | X |
| TXDCB | P4.7 | P9.1 | P7.5 | X |

**CAN IO Pin Selection For XC167 & XC161**

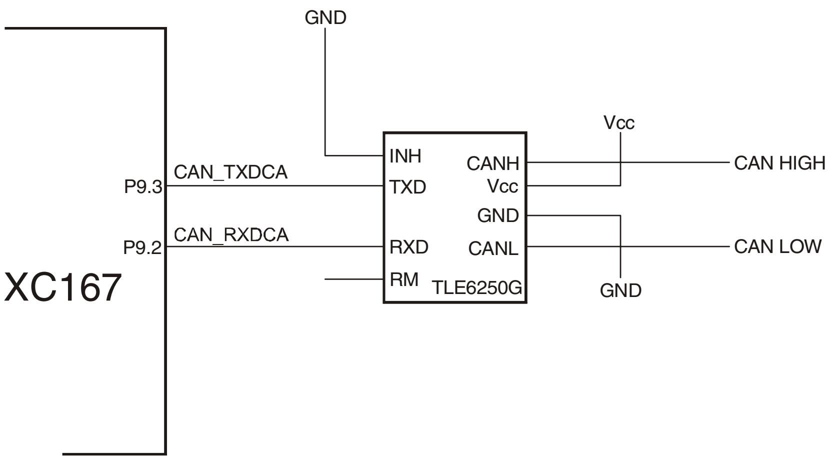
| TwinCAN Signal | Default Pin Set | IO Pin Set 2 | IO Pin Set 3 | |
|---|---|---|---|---|
| RXDCA | P4.5 | P9.2 | P4.7 | |
| TXDCA | P4.6 | P9.3 | X | |
| RXDCB | P4.4 | P9.0 | X | |
| TXDCB | P4.7 | P9.1 | X | |

**CAN IO Pin Selection For XC164CS**

The RXDCx pins to be used are selected through the RISA and RISB fields in the CAN_PISEL register. The TXDCx pins are enabled simply by connecting the required IO pin to the CAN module via the appropriate ALTSEL0Px bit and setting the direction to OUTPUT with the corresponding DPx bit. If Port 9 is used as TXDCx then the corresponding ALTSEL1Px bits must also be set.

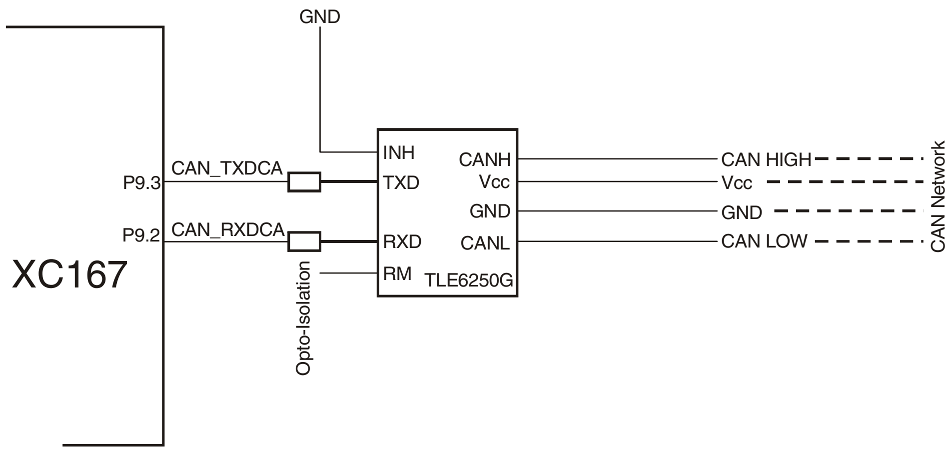
### 9.7.1.2 Interfacing To CAN Transceivers

A common CAN drive chip such as the TLE6250G is attached to the XC167, as shown below:




**Simple TLE6250G CAN Transceiver Interface**

This is a very simple interface and does not provide any significant galvanic isolation between the CAN physical layer and the XC166. It assumes that the all CAN nodes will share a common ground reference, or at least not have any standing voltage offset greater than the TLE6250G's common mode range of –20 to +25v.




**Opto-Isolated CAN Transceiver**

A more robust alternative has opto-isolation between the TLE6250G and the XC167, and assumes that the Vcc and GND for the latter are supplied by two additional wires that run parallel to the CAN data lines. In all cases, for bit rates of above 100kbit/s, it is essential that there is adequate termination on the CAN data lines of 120 ohms if reflections are to be avoided.

P4.0 - General purpose I/O or A16 *or CS3 on XC164*
P4.1 - General purpose I/O or A17 *or CS2 on XC164*
P4.2 - General purpose I/O or A18 *or CS1 on XC164*
P4.3 - General purpose I/O or A19 *or CS0 on XC164*
P4.4 - General purpose I/O or A20 or RxDCB *(CAN module B)*
P4.5 - General purpose I/O or A21 or RxDCA *(CAN module A)*
P4.6 - General purpose I/O or A22 or TxDCA
P4.7 - General purpose I/O or A23 or RxDCA or TxDCA

Port 9, and on the XC161 / 7 Port 7, can optionally be used as the interface to the CAN network. The CAN_PISEL register is used to change the default of RxDCB on P4.4 and RxDCA on P4.5.

# 9.8 Port 5

## 9.8.1 XC166 Analog To Digital Converter

*The information given here is sufficient for the vast majority of XC166 applications that need to make use of the ADC. For applications where ADC performance must be fully characterized or where analog signal sources of high internal resistance and low capacitance are encountered, further work may be required. In these cases you are recommended to read the complete analysis of the peripheral, to be found in Infineon application note AP242801.PDF. You should also consult the datasheet for the XC166 derivative in use.*

The lines of port 5 are a 10, 12 or 16-channel (depending on CPU version), 10-bit or 8-bit resolution successive approximation analog to digital converter input. Alternatively they are general purpose digital input only pins, with Schmitt trigger characteristics. Pins may be allocated to either function freely. To ensure maximum accuracy and freedom from noise, the normal digital IO functionality can be disabled via the P5DIS register so that the port 5 pins are connected solely to the ADC hardware.

### 9.8.1.1 Port 5 Functionality

P5.0 - (all derivatives) Analog input channel 0/Schmitt trigger input 0
.
.
P5.9 - (all derivatives) Analog input channel 9/Schmitt trigger input 9
P5.10 - (XC161/167) Analog input channel 10/ Schmitt trigger input 10/Timer6 direction
P5.11 - (XC161/167) Analog input channel 11/ Schmitt trigger input 10/Timer5 direction
P5.12 - (XC167) Analog input channel 12/ Schmitt trigger input 10/Timer6 count input
P5.13 - (XC167) Analog input channel 13/ Schmitt trigger input 10/ Timer5 count input
P5.14 - (XC167) Analog input channel 14/ Schmitt trigger input 10/Timer4 direction
P5.15 - (XC167) Analog input channel 15/ Schmitt trigger input 15/Timer2 direction

The analog to digital convertor (ADC) is a very high-performance unit with sample-and-hold, auto-calibration and a large number of special conversion modes that are designed to suit real time applications. It has a worst case TUE (Total Unadjusted Error) of +/-2 LSB. Indeed, on several occasions, the XC166 has been selected for applications simply on the quality of the ADC - a case of a great ADC with a free 16 bit microcontroller attached to it!

The XC166 ADC is an enhanced version of that found on the classic C167 family. It can operate though in a "C167 compatibility mode" to allow the easy porting of applications. However any new XC166 designs should use the enhanced mode as it offers some important new facilities such as being able to set different conversion times for normal and injected conversion plus the option of 8 or 10-bit resolutions. It is assumed that enhanced mode will be used in the following.

Given a suitable board layout the ADC can yield a genuine 10-bit resolution, with guaranteed monotonicity (i.e. an increasing input voltage will never result in a smaller digital output). With 4.9mV per bit, robust grounding of the analog ground input plus the provision of guard tracks between signal lines is essential. The analog reference must be a true voltage reference and not just the $V_{SS}$ - see section 9.8.6.

### 9.8.1.2 ADC Conversion Modes (Enhanced)

In addition to the standard single conversion mode, it is possible to set the ADC up to convert a single channel continuously, so that every 2.55us (at 40MHz) a new value will be ready in the ADDAT results register. An interrupt request may be generated to move the data into a RAM buffer but, more usually, the peripheral event controller (PEC) is used to automatically move the result to either a single RAM location or an array. Thus the ADC can collect values into an array with *no CPU intervention,* other than in the latter case, a sub-1us interrupt routine to reset the array pointer `("DTSPx = (unsigned short) _sof_(&ad_store[0])").`

Building on this, in the autoscan mode a number of analog channels can be converted sequentially, with the results being continuously transferred by the PEC into a RAM buffer, so that at any one time the RAM array contains the latest values from each of up to 16 channels, again with minimal CPU activity. One point to bear in mind is that channels that are to be included in the autoscan process must be on adjacent channels, as the mode will convert the channel number that appears in the lower four bits of the ADCON control register first, working in sequence down to channel 0. Thus when choosing what analog signals to attach to which port 5 pin, make sure that any inputs that could benefit from the autoscan conversion are grouped together on the bottommost pins.

Advanced conversion modes are "wait for ADDAT read mode" and "channel injection" mode. The former inhibits further conversions until the last result is read, so that unused conversion data is not accidentally over-written or lost. The channel injection feature is aimed at allowing analog conversions to be made coincident with some event that is asynchronous to the software execution or the normal operation of the convertor. With the ADC being able to automatically scan through a number of channels continuously, making a conversion of a specific channel that is not included in the sequence is taken care of by "Injecting" a conversion by setting the ADCRQ bit. The ADC will finish any conversion that was in progress due to the autoscan mode and make a fresh conversion of the channel specified in the top four bits of ADDAT2 and placing the result in the lower 10- bits of the same register (bits 11-2 in enhanced mode). The autoscan process then resumes. The user must ensure that the wait-for ADDAT-read mode is activated.

The most important use of the injection mode is to make a conversion of a specified channel in response to a level transition on the CC31 port pin (P7.7). Typical examples of where this is useful are the crankshaft-synchronised reading of the inlet manifold pressure in an engine management system. Alternatively, a period match of Timer 13 in CAPCOM6E can trigger an injected conversion so that a reading of the current in the windings of a motor drive at a specific flux angle can be taken.

### 9.8.2 ADC Basic Conversion Clock

The ADC is clocked from the $f_{SYS}$ signal that is identical to the $f_{CPU}$ that drives the XC166 CPU core. It is sometimes referred to as "$f_{ADC}$" and is generally 40MHz. However the ADC hardware is limited to 0.5MHz to 20MHz so the $f_{BC}$ (basic conversion clock) must not be outside this range. In enhanced mode, this is done by setting the ADCTC field of ADC_CTR2 to a minimum of 0x1 so that $f_{BC} <= f_{SYS}/2$. Setting the field to its maximum value of 0x3F (divide by 64) results in a conversion clock of 0.625MHz, still in the acceptable range.

### 9.8.3 ADC Calibration

The ADC is calibrated automatically at each power-on reset. This process takes 660us at $F_{CPU}$ = 20MHz clock cycles. The CAL flag is set for the duration of this process. A post-conversion calibration is performed that takes 12 clock cycles. However this is not essential, ONLY provided the analog reference voltage is guaranteed to have been stable during the power-on reset calibration phase. Disabling the post-conversion calibration (bit ADC_CTR0.CALOFF = 0) will significantly reduce the conversion time.

### 9.8.4 Over-Voltage Protected Analog Inputs

Despite the genuine 10-bit resolution, the XC166's analog inputs can be easily protected against out-of-range voltage inputs as might occur under a fault condition in a real system. Clamping diodes allow a simple series resistor on each analog input to provide a good level of protection against excessive voltage of either polarity.

Unlike many microcontroller A/D converters, the total unadjusted error (TUE) on any input is guaranteed even if an unselected channel has a fault condition voltage of over 5v or under 0v applied to it. Under these conditions, most converters will start to give erroneous readings on other channels, which can have unsafe side-effects as from software, it is very difficult to detect a loss of accuracy. The channel with the fault will read as either 0x0000 or 0x03FF for under- and over-voltages respectively.

The only requirement that must be satisfied to allow the continued correct operation of the fault-free inputs is that the sum total of the fault current flowing into any two unselected analog channels must be less than 10mA. A simple current-limiting resistor can thus prevent the fault affecting other channels.

The series protection resistor (Rap) to be added to the analog inputs can be easily calculated by:

Rap = (Vmax – Vcc)/Imax

Where: Vmax = maximum fault voltage & Imax = maximum permissible current flow

For an automotive application where a common fault condition voltage might be 14v, the series resistor would be around (14v – 5v)/0.010 = 1K0. Of course, this additional resistance will have to be added to the source resistance of the analog signal source itself and it is important to ensure that the sample time is long enough to guarantee a stable voltage on the sample-and-hold capacitor, as outlined in section 9.8.5.

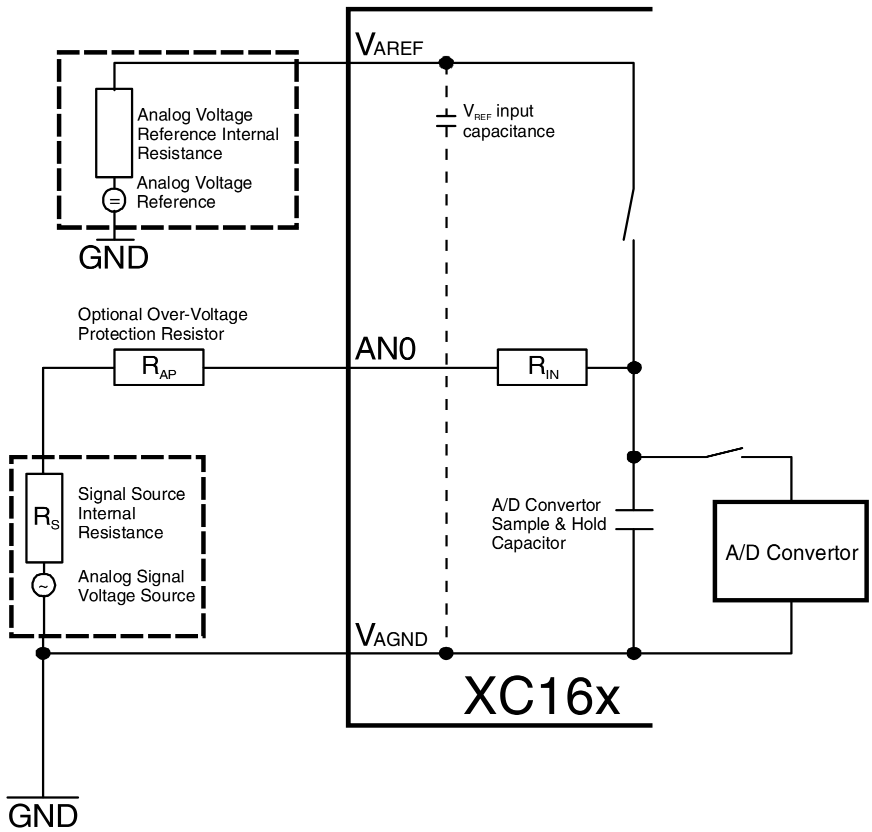
## 9.8.5 Matching The A/D Inputs To Signal Sources

It is possible to alter the apparent input resistance of the analog inputs to allow a better match to the internal resistance of the signal source that is driving them. Sources that change rapidly and that are to be read frequently require a fast conversion time but this will reduce the time available to charge the sample-and-hold (SAH) capacitor[1] in the A/D converter itself.

Thus such signal sources must have a low internal resistance ($R_S$) if the voltage level on the sample-and-hold capacitor is to be fully charged and stable by the time the conversion begins. If the signal can be converted more slowly this requirement is relaxed as the sampling time can be set to a larger value. Thus the internal resistance of the source can be greater without loss of accuracy.

Of course, extending the sampling time does not physically alter the input resistance as it is always several megohms. Also, when a channel is not being read, it has an extremely high impedance with a leakage current of around 10nA. As the sample-and-hold appears to be a simple RC filter whose series resistor is the internal resistance of the source, it is just a matter of making sure that there is sufficient current drive in the signal source to charge up the sampling capacitor before the conversion begins. Provided that $R_S$ is much greater than the typically 250 Ohm series resistor $R_{IN}$ (which in most situations it is), the effects of $R_{IN}$ can be ignored.

The user must also consider the internal resistance of the analog reference voltage source applied to the $V_{AREF}$ pin on the XC166. Again, the reference voltage source must be able to fully charge the input capacitance of this pin within one conversion clock period. The table




**ADC Equivalent Circuit**

shown applies to the situation when the capacitance of the source is greater than the input capacitance of the ADC channel.

The ADCTC and ADSTC bits in the ADC_CON A/D converter control register allow the user to easily alter the rate at which the converter hardware is clocked and thus the length of the sampling time (for SAH capacitor charging) and the conversion phase. The basic timing of the A/D unit is the conversion clock and as the sampling clock is derived from this, the choice of sampling and conversion time is not unlimited. The next table gives the possible legal combinations of conversion and sampling times with the maximum signal and reference internal resistances that are acceptable in each case. This assumes that the capacitance of the lines leading to the analog pin plus that of the source itself is less than 65pF. In practice this figure is likely to be much higher and so it only gives a rough indication of the maximum source resistance. There is a thorough examination of the characteristics of the analog inputs in application note AP2428, available on the Infineon website.

**Default Configuration At $f_{CPU}$ = 40MHz**

| ADC_CTR2.ADCTC = 000001y | ADC_CTR2.ADSTC0 = | 000000 | 000001 | 000011 | 000111 | 001111 | 011111 | 111111 | ADSTC |
|---|---|---|---|---|---|---|---|---|---|
| Sample Time (us) | | 0.2 | 0.4 | 0.8 | 1.6 | 3.2 | 6.4 | 12.8 | us |
| Overall Conversion Time (us) | (With post calibration) | $\boxed{2.95}$ | 3.15 | 3.55 | 4.35 | 5.95 | 9.15 | 15.55 | us |
| | (Without post calibration) | 2.35 | 2.55 | 2.95 | 3.75 | 5.35 | 8.55 | 14.95 | us |
| Varef Source Resistance (Ohms) | | 53 | 53 | 53 | 53 | 53 | 53 | 53 | Ohms |
| Signal Source Resistance (Ohms) | | 624 | 1499 | 3247 | 6745 | 13740 | 27730 | 55709 | Ohms |
| **ADC_CTR2.ADCTC = 000011y** | **ADC_CTR2.ADSTC0 =** | **000000** | **000001** | **000011** | **000111** | **001111** | **011111** | **111111** | **ADSTC** |
| Sample Time (us) | | 0.4 | 0.2 | 0.4 | 0.8 | 6.4 | 12.8 | 25.6 | us |
| Overall Conversion Time (us) | (With post calibration) | 5.75 | 5.55 | 5.75 | 6.15 | 11.75 | 18.15 | 30.95 | us |
| | (Without post calibration) | 4.55 | 4.35 | 4.55 | 4.95 | 10.55 | 16.95 | 29.75 | us |
| Varef Source Resistance (Ohms) | | 356 | 356 | 356 | 356 | 356 | 356 | 356 | Ohms |
| Signal Source Resistance (Ohms) | | 1499 | 624 | 1499 | 3247 | 27730 | 55709 | 111668 | Ohms |
| **ADC_CTR2.ADCTC = 000111y** | **ADC_CTR2.ADSTC0 =** | **000000** | **000001** | **000011** | **000111** | **001111** | **011111** | **111111** | **ADSTC** |
| Sample Time (us) | | 0.8 | 0.4 | 0.8 | 1.6 | 12.8 | 25.6 | 51.2 | us |
| Overall Conversion Time (us) | (With post calibration) | 11.35 | 10.95 | 11.35 | 12.15 | 23.35 | 36.15 | 61.75 | us |
| | (Without post calibration) | 8.95 | 8.55 | 8.95 | 9.75 | 20.95 | 33.75 | 59.35 | us |
| Varef Source Resistance (Ohms) | | 962 | 962 | 962 | 962 | 962 | 962 | 962 | Ohms |
| Signal Source Resistance (Ohms) | | 3247 | 1499 | 3247 | 6745 | 55709 | 111668 | 223586 | Ohms |
| **ADC_CTR2.ADCTC = 001111y** | **ADC_CTR2.ADSTC0 =** | **000000** | **000001** | **000011** | **000111** | **001111** | **011111** | **111111** | **ADSTC** |
| Sample Time (us) | | 1.6 | 3.2 | 6.4 | 12.8 | 25.6 | 51.2 | 102.4 | us |
| Overall Conversion Time (us) | (With post calibration) | 22.55 | 24.15 | 27.35 | 33.75 | 46.55 | 72.15 | 123.35 | us |
| | (Without post calibration) | 17.75 | 19.35 | 22.55 | 28.95 | 41.75 | 67.35 | 118.55 | us |
| Varef Source Resistance (Ohms) | | 2174 | 2174 | 2174 | 2174 | 2174 | 2174 | 2174 | Ohms |
| Signal Source Resistance (Ohms) | | 6745 | 13740 | 27730 | 55709 | 111668 | 223586 | 447423 | Ohms |
| **ADC_CTR2.ADCTC = 011111y** | **ADC_CTR2.ADSTC0 =** | **000000** | **000001** | **000011** | **000111** | **001111** | **011111** | **111111** | **ADSTC** |
| Sample Time (us) | | 3.2 | 6.4 | 12.8 | 25.6 | 51.2 | 102.4 | 204.8 | us |
| Overall Conversion Time (us) | (With post calibration) | 44.95 | 48.15 | 54.55 | 67.35 | 92.95 | 144.15 | 246.55 | us |
| | (Without post calibration) | 35.35 | 38.55 | 44.95 | 57.75 | 83.35 | 134.55 | 236.95 | us |
| Varef Source Resistance (Ohms) | | 4598 | 4598 | 4598 | 4598 | 4598 | 4598 | 4598 | Ohms |
| Signal Source Resistance (Ohms) | | 13740 | 27730 | 55709 | 111668 | 223586 | 447423 | 895095 | Ohms |
| **ADC_CTR2.ADCTC = 111111y** | **ADC_CTR2.ADSTC0 =** | **000000** | **000001** | **000011** | **000111** | **001111** | **011111** | **111111** | **ADSTC** |
| Sample Time (us) | | 6.4 | 12.8 | 25.6 | 51.2 | 102.4 | 204.8 | 409.6 | us |
| Overall Conversion Time (us) | (With post calibration) | 89.75 | 96.15 | 108.95 | 134.55 | 185.75 | 288.15 | 492.95 | us |
| | (Without post calibration) | 70.55 | 76.95 | 89.75 | 115.35 | 166.55 | 268.95 | 473.75 | us |
| Varef Source Resistance (Ohms) | | 9447 | 9447 | 9447 | 9447 | 9447 | 9447 | 9447 | Ohms |
| Signal Source Resistance (Ohms) | | 27730 | 55709 | 111668 | 223586 | 447423 | 895095 | 1790441 | Ohms |

**Tuning The Analog To Digital Converter Effective Input Impedance**

It can be seen that at the (default) fastest combined sampling and conversion time of 2.95us (post calibration enabled), the signal source resistance must be less than 624 Ohms to ensure complete charging of the sampling capacitor. At the other extreme, with a sampling time of 409.6us (resulting in an overall conversion time of 492.95us), the source resistance can be up to 1.8MOhm. If a series protection resistor is being used, the figure in the table for the signal source resistance must be reduced by the resistor's value as it is effectively in series with the source's own internal resistance.

As these timing characteristics are programmable on-the-fly in software, it is entirely possible to make special settings to the ADCTC and ADSTC bits prior to the conversion of any channel which has a much higher internal source resistance than the others.

[1] *In reality this is not a single capacitor but a series of 8 or 10 parallel capacitors (one per bit of resolution) that are binary-weighted so that each one is double the size of the previous one. By connecting various combinations of these charged capacitors to a comparator, the digital value of the input voltage is determined.*

### 9.8.6 Analog Reference Voltage

Any external voltage reference used to drive $V_{AREF}$ must have sufficient strength to remain stable during the charging of the ADC's internal capacitor network during the conversion phase. The duration of this is related to the chosen conversion time and resolution, with 10 bit conversions taking longer than 8-bit. In most situations, a 100nF capacitor connected close to the $V_{AREF}$ and $V_{AGND}$ pins is sufficient. Also, any series resistance between the reference source and the $V_{AREF}$ must be less than 50 Ohms.

## 9.9 Board Design Issues

At 10 bits, each bit is around 5mV so very great care is required to get any meaningful information from the LSB! Here are some guidelines for laying out PCBs for best analog performance.

### 9.9.1 Component Placement

Try to group all analog components grounded together in one area and all digital components in another. Common power supply-related components should be centrally located. Mixed signal components, including the XC166 itself, should be located between the analog and digital zones, with only analog pins in the analog area and digital pins in the digital area! Rotating the CPU can often make this easier.

### 9.9.2 Power Supply

Place the analog power and voltage reference regulators over the analog plane. The same goes for the digital power regulators. Analog power traces should be over the analog ground plane. The same holds for the digital power traces. Decoupling capacitors should be close to the microcontroller pins, or positioned for the shortest connection to the pins, with wide traces to reduce impedance. If both large electrolytic and small ceramic capacitors are recommended, make the small ceramic capacitor closest to the microcontroller pins.

Make sure that the analog reference voltage is not able to be on whilst the CPU's $V_{DD}$ is off otherwise the ESD (electrostatic discharge) protection diodes inside the XC166 will have to power the entire CPU, which is highly undesirable.

### 9.9.3 Ground Planes

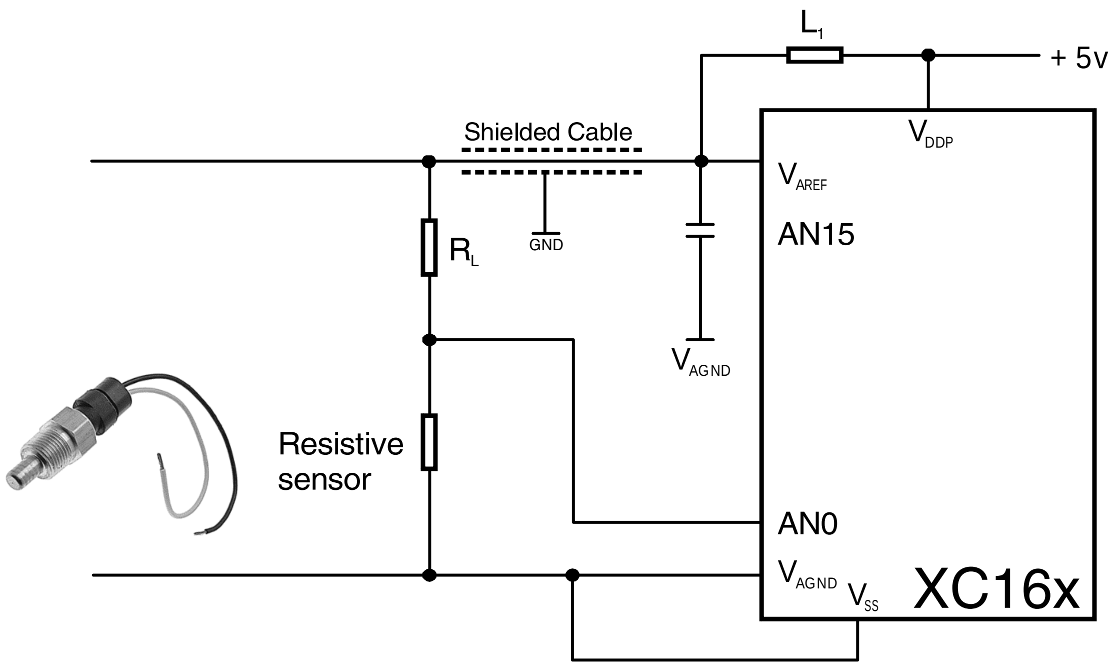
It is best to provide separate analog and digital ground planes on the same layer separated by a gap, with the digital components over the digital ground plane and the analog components over the analog ground plane. Analog and digital ground planes should only be connected at one point in most cases, the best place being below the microcontroller. Have vias available in the board to allow alternative points.

The connection between the analog and digital grounds should be near to the power supply or near to the power supply connections to the board. Alternatively it should be near to the CPU. For boards with more than 2 layers, do not overlap analog and digital-related planes. Do not have a plane that crosses the gap between the analog ground plane and the digital ground plane region.

## 9.10 Connecting The ADC To Signal Sources

There are three main methods of connecting the ADC to external signal sources. Each has its advantages and disadvantages and these will be examined here.

### 9.10.1 Ratiometric Mode




**ADC Connections For Ratiometric Mode**

This is useful where the inputs are resistive sensors (thermistors, strain gauges etc.) or where the output of the sensor is proportional to the measured quantity and the sensor supply voltage. The 5v supply for the XC166 is connected to the $V_{AREF}$ pin so that the ADC reference voltage is in fact the peripheral supply voltage. The 5v and $V_{AGND}$ are then used to power all the analog sensors, which in some applications might represent a considerable length of cabling off the XC166 board. The sensor outputs (usually in the form of potential dividers) are then connected to the XC166's analog pins. Thus the ADC measures the ratio of the sensor resistance to the supply voltage.

The advantage of this scheme is that if the 5v supply changes, both the ADC and the sensors are subject to the same change and so as far as the digital result is concerned, the ratio of the signal to the analog reference voltage is unaltered.

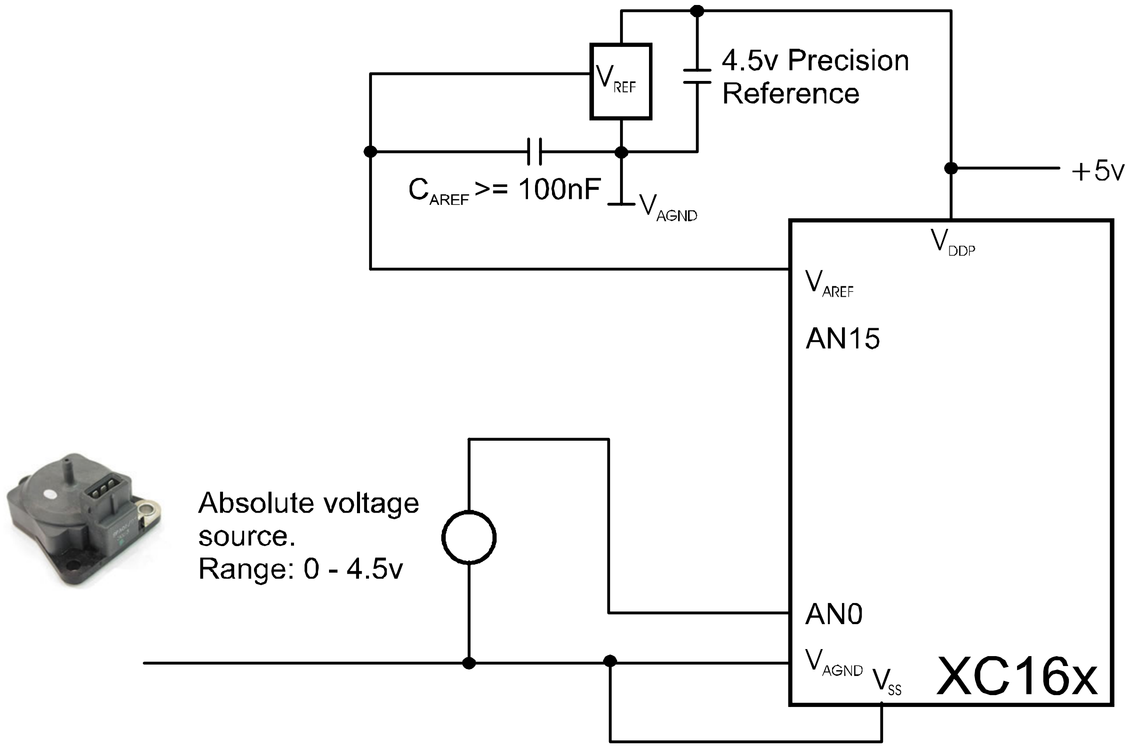
#### 9.10.1.1 Ratiometric Advantages

*- Cheap*
*- Accurate*
*- Noise resistant*
*- Full 0-5v analog conversion range possible*

#### 9.10.1.2 Ratiometric Disadvantages

*- CPU 5v supply must be connected to a considerable length of wire so $V_{DD}$ may be upset by noise or other disturbances.*
*- Unsuitable for reading inputs where the absolute voltage is significant*

### 9.10.2 Fixed Precision Reference




**Fixed Precision Reference Mode**

This method allows very accurate measurements of absolute voltages by driving the $V_{AREF}$ pin with an independent voltage reference. Unfortunately as the XC166 peripheral supply voltage is nominally 5v, it is not always possible to use a 5v reference as they usually require at least a 5.2v supply. In practice the highest reference is likely to be 4.5v, which gives 4.4mV per bit at 10-bit resolution. However the measurable analog voltage range is then limited to 0v - 4.5v.

This method can also be used for high accuracy measurements using the ratiometric principle by driving sensors from the 4.5v reference, although care should be taken that it has sufficient current sourcing ability (see above).

The $V_{AREF}$ pin must be set between 4.5v and $V_{DDP}$ + 0.2v (i.e. 5.2v) if the best TUE of +/- 2LSB is to be achieved. If it is in the range 4v <= $V_{AREF}$ < 4.5v, the TUE is increased to +/- 3 LSB. This effectively rules out using voltage references of less than 4.5v.

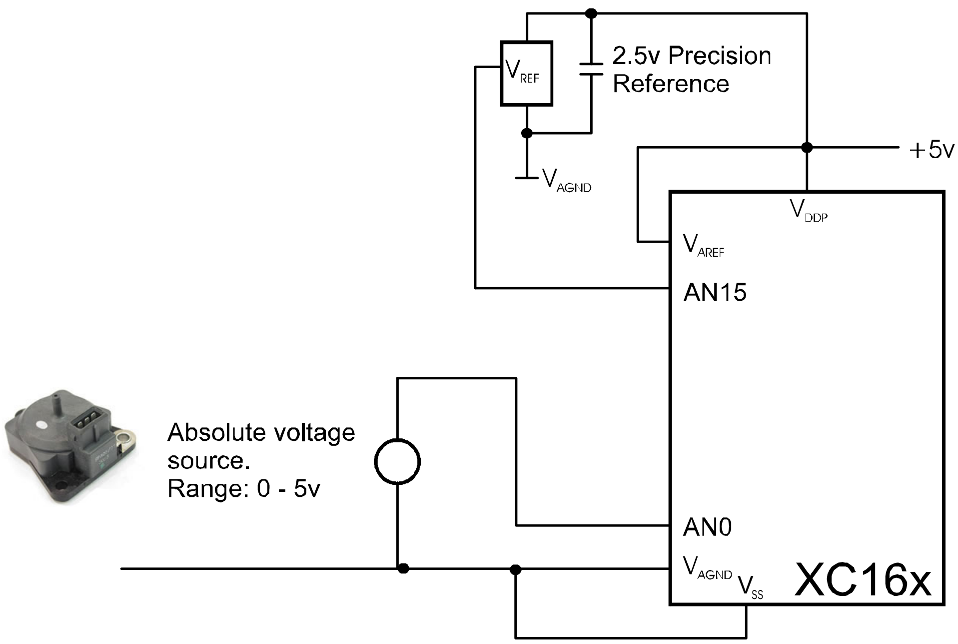
#### 9.10.2.1 Fixed Precision Reference Advantages

*- Very accurate conversions*
*- Low noise*

### 9.10.2.2 Fixed Precision Reference Disadvantages

*- Cost of voltage reference*
*- Reduced dynamic range*

## 9.10.3 Corrected Conversion Mode




**Corrected/Compensated Mode**

This method uses the 5v to drive the $V_{AREF}$ pin, as in the simple ratiometric case but here a voltage reference is used to correct the readings mathematically. Thus the full 0 – 5 v input range is available. Any channel that must be measured to a particularly high degree of accuracy is scaled using a reading from the fixed voltage reference, here permanently connected to channel AN15.

For a 2.5v reference,

```
Corrected reading = actual reading * 511/(AN15 reading)
```

### 9.10.3.1 Corrected Conversion Advantages

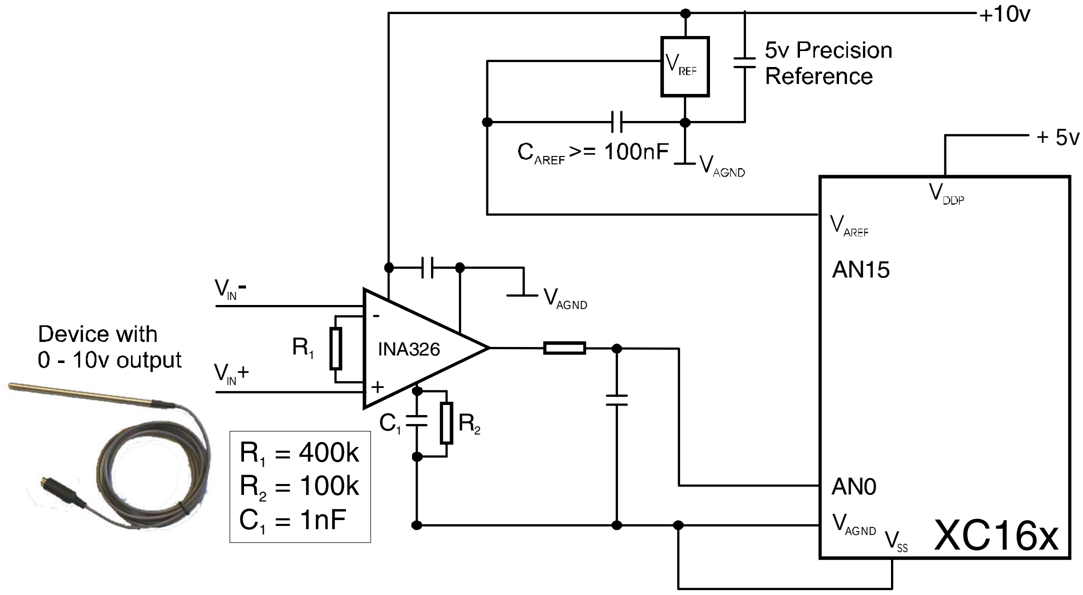
*- Very accurate conversions*
*- Low noise*

### 9.10.3.2 Corrected Conversion Disadvantages

*- Cost of voltage reference*
*- Small processing overhead*

## 9.10.4 Interfacing To Analog Voltages Greater Than 5v

In industrial control applications, it is very common to have to interface from sensors or other devices that use 0-10v range. As the maximum $V_{AREF}$ is the 5v supply, some sort of scaling is required. As sensors are likely to be some distance from the CPU, possibly several metres away, they are often on a different power supply. Ground reference is then likely to be a problem so a simple 0-5v input is inadequate. Here, a buffer with a differential input and a gain of 0.5 is required. In practice some sort of instrumentation amplifier will be needed to protect the XC166 from any common mode voltages (voltage mismatches between two different ground references) and to reduce the 0-10v range to 0-5v.




**10v Input Signal Conditioning Scheme**

## 9.11 Port 6

**This port is not present on the XC164.**

General purpose bi-directional I/O port with push-pull or open drain outputs. If an external bus is required, part of P6 can be used for chip select signals for memory decoding and external device enabling. The number of pins to be used as chip selects is set by the P0 configuration resistors and / or software - see section 1.5. The alternate function of the upper nibble on P6 is the Hold, Hold acknowledge and Bus request signals.

P6.0 - Port 6.0 / Chip select 0
P6.1 - Port 6.1 / Chip select 1
P6.2 - Port 6.2 / Chip select 2
P6.3 - Port 6.3 / Chip select 3
P6.4 - Port 6.4 / Chip select 4
P6.5 - Port 6.5 / HOLD
P6.6 - Port 6.6 / HLDA
P6.7 - Port 6.7 / BREQ

## 9.12 Port 7

This port is not present on the XC164.

General purpose bi-directional I/O port with push-pull or open-drain outputs. Also input/output pins for second capture compare unit, channels 28 to 31. Port 7 can also be optionally used for the CAN signals.

P7.4 - Port 7.4 / CAPCOM (unit 2) channel 28 / RxDCB
P7.5 - Port 7.5 / CAPCOM (unit 2) channel 29 / TxDCB
P7.6 - Port 7.6 / CAPCOM (unit 2) channel 30 / RxDCA
P7.7 - Port 7.7 / CAPCOM (unit 2) channel 31 / TxDCA

## 9.13 Port 9

General purpose bi-directional I/O port with push-pull or open drain outputs. Also input/ouput pins for second capture compare unit, channels 16 to 21. Port 9 can also be optionally used for the CAN signals, provided the appropriate ALTSEL1P9 bit is set to 1 for any TXDCx signal.

P9.0 - Port 9.0 / CAPCOM (unit 2) channel 16 / RxDCB
P9.1 - Port 9.1 / CAPCOM (unit 2) channel 17 / TxDCB
P9.2 - Port 9.2 / CAPCOM (unit 2) channel 18 / RxDCA
P9.3 - Port 9.3 / CAPCOM (unit 2) channel 19 / TxDCA
P9.4 - Port 9.4 / CAPCOM (unit 2) channel 20
P9.5 - Port 9.5 / CAPCOM (unit 2) channel 21

## 9.14 Port 20

As it is intended predominantly for single chip applications, signals such as RD and WR are present on Port 20. In this way as many pins as possible will be available as general purpose bi-directional I/O port.

P20.0 - Port 20.0 / RD
P20.1 - Port 20.1 / WR
*P20.2 - Port 20.2 / READY (only on XC161 / XC167)*
P20.4 - Port 20.4 / ALE
**P20.5 - Port 20.5 / EA**
P20.12 - Port 20.12 / RSTOUT

The /EA pin is only read by the CPU when coming out of reset and is thereafter ignored. Therefore this can be used as general IO, if the application is short of pins. It is thus recommended that EA is connected to +5v via a 10k resistor. This will also assist when using a full XC166ED (emulation device) in-circuit emulator like the DPROBEXC.

# 9.15 Summary Of Port Pin Interrupt Capabilities

## 9.15.1 Interrupts From Port Pins

The XC166 family can generate interrupts from dual-function port pins on rising, falling or both edges. For example, the T2IN pin can be used as a means to generate interrupts in response to incoming edges, completely neglecting any functions it has that are associated with Timer 2. The same is true for T4IN, T3IN, T5IN and T6IN. The CAPCOM pins (CCxIO) can also be used for simple edge-triggered interrupts. The pins are scanned every 200ns (40MHz clock) so it will take a *maximum* of 200ns for the XC166 to detect the interrupt request. Port 2.8-2.15 (Port 1H.0 – Port 1H.7 on XC164) provides 8 fast interrupt pins that are scanned every *25ns* so the detection time is 25ns. The latency times of 9 to 13 state times (225 - 325ns) must be added to these for the total time from an edge arriving at a pin to the interrupt vector being executed.

The 144 pin XC161 / XC167 can generate interrupts from up to 37 pins (15 for XC164), depending on the bus mode being used.

# 10 Typical XC166 Family Applications

Here are some applications in which we know the 166 family is being used. In almost every case, the family was selected for one or more of the following reasons:

Very high processing performance in C
Large number of interrupt pins
High resolution PWM generators
PEC DMA controller
Close coupled core and peripherals
Low EMC emissions
Very high reliability high temperature FLASH
FLASH management features
Large number of IO pins
Up to 32 capture and compare pins
Part 2.0B CAN peripheral
No microcoded TPU!
Very low current consumption per MIP
Some variants are second-sourced
CAPCOM6E motor drive peripheral

*Note: In cases where there were specific reasons for selection that we know of, they are given.*

## 10.1 Automotive Applications

**Formula One engine management and gearbox control systems -** *High CPU performance allowed innovative control algorithms. Close coupled CPU with 16 channel CAPCOM unit. Ease using BREQ/HOLD/HOLDA to share common RAM in dual processor system.*
**Indy car engine management systems -** *Entire program could be in C language without losing performance, including high speed interrupt sections - previous project abandoned due to difficulty in altering TPU programming in CISC CPU. Availability of part 2.0B CAN peripheral. Quality of development tools. High level of support from Hitex.*
**Touring car engine management -** *Ease of programming as entire program in C. Close coupling of CAPCOM to CPU simplified program design. Deterministic interrupt latency times.*
**Low-volume prestige car engine management system -** *Ease of programming as entire program in C. Close coupling of CAPCOM to CPU simplified program design. Previous project compromised by difficulty in applying TPU. Part 2.0B CAN interface.*
**Competition ignition systems**
**Diesel unit injector control**
**Anti-lock braking systems -** *Large number of frequency-measuring inputs, high CPU performance, deterministic interrupt response, high resolution PWM unit, part 2.0B CAN peripheral.*
**Diesel injection pump control -** *Ease of programming an entire program in C. Close coupling of CAPCOM to CPU simplified program design. Part 2.0B CAN interface. Second sourced part. Very high CPU performance and I/O pin count, flexible bus interface, in-circuit reprogrammability via bootstrap loader, low cost in volume, high integration, low power consumption, quality of development tools.. High level of support from Hitex.*
**Petrol engine management systems**
**Marine diesel engine regulators -** *Flexibility of peripherals, outright CPU performance, ease of programming in C, I/O pin count, flexible bus interface, part 2.0B CAN peripheral, compatible with very low cost versions like C161.*
**Active suspension control**
**Electronically-assisted power steering controller -** *Near DSP performance, CAPCOM6E motor driver, high reliability FLASH, high I/O pin count, 32 channels of capture and compare, part 2.0B CAN, ease of programming, second sourced part.*

## 10.2 Industrial Control Applications

**AC induction motor drives (vector control)** - *Near DSP performance at low cost, flexible sinewave synthesis via CAPCOM unit, full vector control possible.*
**AC induction motor drives (open loop)** - *Low cost, high integration, ease of sinewave synthesis via CAPCOM, in-circuit reprogrammability of FLASH EPROM.*
**Linear induction motor control** - *Easy waveform generation via CAPCOM unit.*
**DC brushless motor control** - *High CPU performance, simple commutation via angle-driven CAPCOM unit, non-intrusive PEC update of switching points.*
**C Programmable logic controllers (PLC)** - *High performance in C, simplicity of bus design, ease of interfacing to LCD panels, low CPU cost.*
**High speed packaging machines** - *Very high CPU and CAPCOM performance, peripheral event controller allows non-intrusive drive of CAPCOM, low cost for performance, high I/O pin count.*
**Bottling line barcode printers** - *Very fast conversion of ASCII text to bitmap images using C, non-intrusive PEC transfer of image data to inkjet printhead.*
**Cigarette rolling and packaging machines** - *Very high CPU performance, ease of coupling CAPCOM to rotating shafts, ease of coupling CAPCOM to solenoids, high resolution PWM module, part 2.0B CAN.*
**Printing press controls** -*Multi-channel pulse measurement and generation via CAPCOM, automatic angle-to-time domain conversion in CAPCOM, part 2.0B CAN, very high CPU performance, PWM module.*
**Cotton carding machine controls**
**Power inverter controllers**
**Elevator controls**
**Generator protection**
**Networked security systems**
*Easy creation of 8 software UARTs via CAPCOM, part 2.0B CAN peripheral, high quality development tools.*
**Intelligent CCTV security system** - *Real time synchronisation to lines, 1us sampling and PEC transfer of data into RAM array, very high CPU performance.*
**22kW UV lamp controller in printing press** – *CAN, number of interrupt pins, suitability for safety-critical applications, on-chip FLASH*

## 10.3 Telecommunications Applications

**Modem concentrators** - *Easy upgrade from 8032, large address space, fast context switch, easy multi-tasking, low cost; UART, ease of implementing software UARTs.*
**ISDN terminal equipment** - *5x higher performance than 16-bit 8051 at same price, high pin count, compatibility of C compiler to C51.*
**Mobile radio base stations** - *High I/O pin count, low EMC emissions.*
**GSM cellphone handsets** - *Low current consumption, fast context switch.*
**ISDN test gear** - *Easy 33 bit period measurement, high CPU performance, low current consumption-per-instruction-per-second.*
**Internet server cooling supervisors** - *High accuracy A/D converter, ability to drive 4x three-phase motors from dual CAPCOM unit.*
**Profibus interfaces**
**CAN to PC interfaces** - *Easy implementation of master-slave ISA bus interface, part 2.0B CAN peripheral.*
**PCMCIA CAN interface card** - *Very small package size, low power consumption, very high CPU performance, UART.*

### 10.3.1 Transport Applications

**Marine radar systems** - *Very high integration, very high CPU performance allows tracking of 24 targets, easy interface to VGA graphics, easy frequency lock to GPS markers, lower cost family members available.*
**Marine positioning and navigation systems** - *Easy upgrade from 8032, very good floating point performance in C, quality of development tools.*
**Networked traffic signal controllers**
**Bus ticketing systems**
**Taxi meters**

## 10.4 Consumer Applications

**Lighting desk controls -** *Easy 250kbit/s UART, high I/O pin count, 28 PWM channels on CAPCOM, ease of programming in C, 16 channel A/D convertor.*
**Audio mixing desks**
**Video recorder servo controller**
**Hard disk drive controllers -** *Deterministic interrupt latency, high CPU performance, interrupt structure, low cost.*

**TV test gear and pattern generators -** *ease of synchronising to line sync. pulses and pulse generation via CAPCOM. High CPU performance allows useful processing within line.*
**TV mixing desks**
**TV standards converters -** *ease of synchronising to line sync. pulses. PEC capture of incoming lines to array and high CPU performance.*

## 10.5 Instrumentation Applications

**Hand held vibration analyser (battery powered)** - *very low current consumption, high throughput per milliamp, bootstrap loading of FLASH program.*
**Hand held non-destructive testers (battery powered)** - *high performance per milliamp, 33 period measurement with GPT2, SPI via synchronous port.*
**Scanning electron microscopes**
**High voltage precision power supplies -** *low cost, ease of interfacing to large memory areas, simultaneous mixed 8- and 16-bit busses.*

**PCMCIA modem interface (165 inside card)**
**PCMCIA CAN interface (165 inside card)** - *high CPU performance required to process 1MB/s CAN data.*
**Inkjet printer controller** - *high CPU performance for fast bitmap imaging, PEC transfer to synchronous port, low cost.*
**Hand held sound level meters (battery powered)** - *high performance for power consumed, accurate A/D converter, on-chip FLASH EPROM is in-circuit programmable.*

## 10.6 High Integrity, Aerospace, Medical

**Intravenous drug metering systems** – *life-critical medical use of XC16x allowed by Infineon, "proof in use" tested C compiler*
**Peristaltic pumps**
**Heart monitors** - *life-critical medical use of XC16x allowed by Infineon, "proof in use" tested C compiler*
**Aircraft power bus management via CAN** - *Stable long term availability, DO178B-capable testing tools, CAN, real time performance*

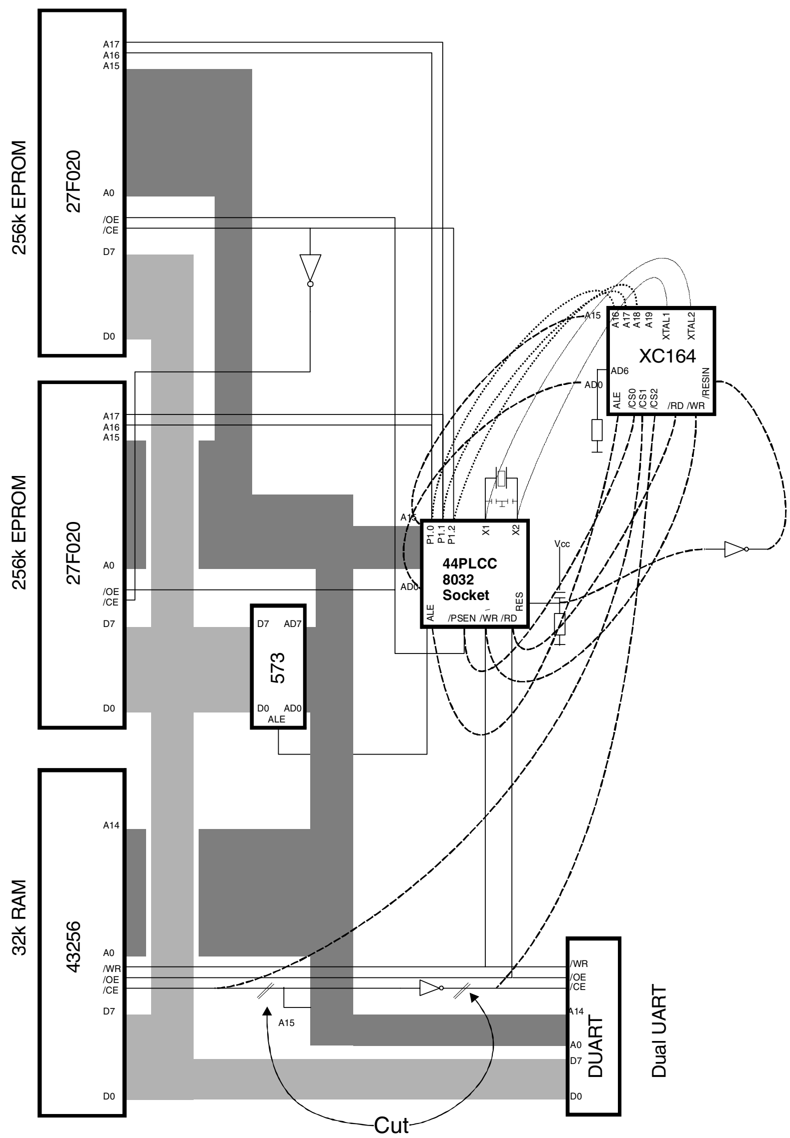
**Military instrumentation & display** - *Stable long term availability, CAN, real time performance*
**Aircraft instrumentation** - *Stable long term availability, DO178B-capable testing tools, CAN, real time performance*
**Unmanned aircraft guidance system** - *Stable long term availability, DO178B-capable testing tools, CAN, real time performance*

**Note:** a quality management report is available for the Keil C166 compiler. This is intended for software quality regimes where the C compiler behaviour must be documented. See www.infineon.com/xc166-family.

# 11 XC166 Compatibility With Other Architectures

The XC166 has an original RISC-like core design that is derived from the popular C167 architecture. It is not related to any older architecture such as 8086 or 6800. This means that it is not possible to execute, for example, 8051 binary code directly. There is a code translator utility available that will take in 8051 assembly programs and emit A166 source files. However, the fact that the most popular 8051 C compiler manufacturer also produces one for the C166, means that the port to the XC166 is not particularly difficult if the program is in C. Of course the peripherals are different but they do have some similarities, which at least makes the job feasible.

The bus interface of the XC166 and 8051 can be quite different but if the 8-bit multiplexed or non-multiplexed modes are used, it is surprisingly easy to hook an XC164 into an 8032 design. In fact the resulting design may be simpler due to the elimination of the address latch which is redundant due to the XC164's non-multiplexed bus. The following shows how an XC164 can be literally wired into an 8032 socket - further details on this are available from Hitex. This was a complex case as the old 8032 design had been stretched over the years to add more and more EPROM, using bank-switching. The change had to be made to 16-bits as the 8032 simply could not execute the vast amount of software fast enough! The XC164 version was some 20 times faster, even with an 8-bit bus...




**Seriously, somebody actually did this….**

# 12 Mounting XC166 Family Devices On PCB's

## 12.1 Package Types

Like most modern microprocessors, all XC devices are in packages that are intended for direct surface mounting onto the PCB. The pin pitch is 0.5mm, with pin counts of up to 144. The increasing number of package types being used for XC166 family devices are listed in the following table:

### 12.1.1 Table Of Common XC166 Derivatives

| Description | FLASH/ROM, PSRAM / DSRAM / DPRAM | Peripherals | Package |
|---|---|---|---|
| XC161CJ-16FxxF | 128k, 2k/4k/2k | 12-Channel 8/10Bit A/D Converter, GPT with 5 Timers, 2 UARTS, 2xSSC, TwinCAN, **SDLM**, I²C, RTC, Power Down Modes, Watchdog, Oscillator Watchdog, 99 GPIO, OCDS | P-TQFP-144 |
| XC161CS-32FxxF | 256k, 2k/4k/2k | 12-Channel 8/10Bit A/D Converter, GPT with 5 Timers, 2 UARTS, 2xSSC, TwinCAN, I²C, RTC, Power Down Modes, Watchdog, Oscillator Watchdog, 99 GPIO, OCDS | P-TQFP-144 |
| XC164CS-16FxxF | 128k, 2k/2k/2k | 14-Channel 8/10Bit A/D Converter, 2x16-CAPCOM Units, **CAPCOM6E**, I²C Bus, Power Down Modes, Watchdog, 2 UARTS, 2xSPI, TwinCAN , 79 GPIO, OCDS | P-TQFP-100 |
| XC164 CS-8FxxF | 64k, 2k/2k/2k | 14-Channel 8/10-bit A/D Converter, 2x16-CAPCOM Units, **CAPCOM6E**, I²C Bus, Power Down Modes, Watchdog, 2 UARTS, 2xSPI, TwinCAN , 79 GPIO, OCDS | P-TQFP-100 |
| XC167CI-16FxxF | 128k, 2k/4k/2k | 16-Channel 8/10Bit A/D Converter, 2x16-CAPCOM Units, **CAPCOM6E** , I²C Bus, Power Down Modes, RTC, 2 UARTS, 2xSPI, TwinCAN , 103 GPIO, OCDS | P-TQFP-144 |

## 12.2 Connecting Emulators To XC166 Family Devices

### 12.2.1 Socketed Devices

In the past, first prototypes would have had the CPU fitted in a socket so that it could be easily replaced after accidents and an emulator could be fitted directly. DIL and PLCC sockets are cheap and readily available. Unfortunately the sockets for TQFP are relatively expensive and not always easy to find. For building development boards they are ideal, as they have the same footprint as the CPUs themselves, so that no board changes are required to fit them. It is advisable to leave 0.2" around the perimeter of the CPU pads as the sockets are somewhat bulkier than the chips.

The socket is an assembly of a base platform with fine contacts around the edge and a clamping ring. There are extensions in the corners with threaded holes to allow the CPU retaining ring to be firmly screwed down. Somewhat confusingly, the Yamaichi sockets are supplied assembled "upside down", so that pegs intended to locate the CPU appear to be positioning studs, designed to fit into holes in the PCB. THIS IS NOT THE CASE - the underside of the CPU platform is flat! The retaining ring must removed to get the real picture. The shape of

the contacts is such that it is very difficult to solder them down using even a very fine soldering iron. Solder paste and a hot-air gun are much more likely to be successful.

Yamaichi are the major supplier of these sockets, but local distributors are usually only interested in bulk orders so a request for ones and twos will not get an enthusiastic response! Hitex keeps a small stock of all Yamaichi socket types for emergencies but we have to charge a higher price for them than component specialists.

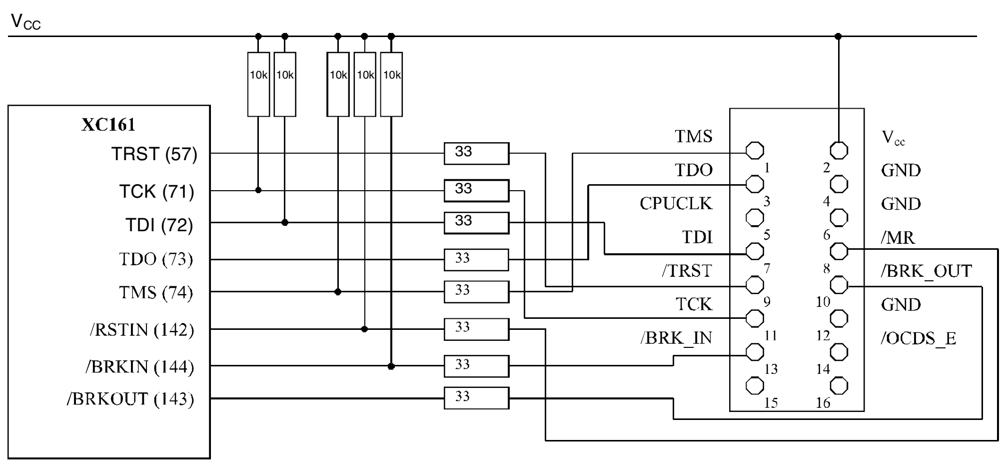
### 12.2.2 Debugging XC Family Applications

The XC family is possibly unique in that there are two different debugging systems offering different levels of functionality. As with most modern architectures, the XC design includes a JTAG interface. The On Chip Debug Support (OCDS) is dubbed a level 1 debugger. It means that a hardware 'wiggler' (an interface between the host PC and the processors JTAG pins) can be connected to the 'standard' target hardware and basic debugging can be performed.

#### 12.2.2.1 XC166 Bondout-Based In-Circuit Emulator

However unlike other architectures that have JTAG interfaces, the XC family also has a full in-circuit emulator (level 2) available. The Hitex DPROBEXC is such a machine. Rather than producing a "bondout" processor as they did for the 'Classic' C167 family, Infineon have produced an Emulation Device ("XC166ED") by using a production version of the silicon and mounting it onto a carrier to add the debugging functionality. The advantages of the Emulation Device are that they follow the stepping of the production silicon with the same 'features' and timings. Although the emulation device has the internal FLASH, the emulators also have 'FLASH Overlay' memory which means that code can be loaded into the emulator's memory which is quicker than programming the FLASH as the JTAG debuggers must. The other main advantage of the full emulator is that there is an option for a full code and data trace and coverage so as far as debugging is concerned, the XC has the best of both worlds – a JTAG debugger for less critical applications and a full in circuit emulator for safety critical applications.

Connecting the level 1 debuggers involves little more than connecting the JTAG pins on the processor to the connector. The 'standard' Infineon connector is a 16 way 0.1" header, but there is no reason this can't be replaced by a connector of your choice. A typical connection scheme is as follows -




**JTAG Connections**

It should be noted that although on the XC161 & XC167, the JTAG pins are dedicated, on the XC164 the JTAG signals are multiplexed with other signals and this must be taken into account during the design. This means that special measures need to be taken if an XC164 design is to be used with a JTAG debugger otherwise only the level 2 emulator-type tool can be used. As well as being used as a debugger, the OCDS system can be used to production program the FLASH memory.

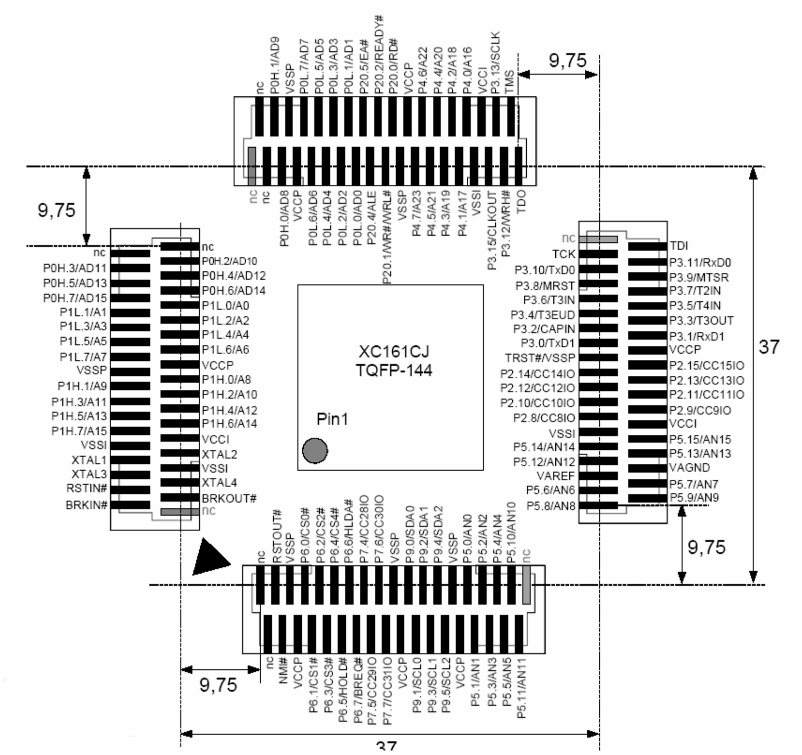
### 12.2.2.2 JTAG Connectors

The 0.1" JTAG connector is just a standard 16-way unboxed pin header.  There are right-angle versions readily available too.    The 0.050" connector type is usually a Samtec part, “TFM-110-32-SD”.

## 12.3 Connecting An In-Circuit Emulator

There are four methods of connecting the level 2 debugger, a.k.a. in-circuit emulator, covered below.

### 12.3.1 The QuadConnect




If sufficient space can be afforded on the target board, the recommendation would be to use the quad connect scheme. This involves fitting four connectors around the processor on a 0.050 pitch which connect to every processor pin. An adaptor board plugs on to the quad connectors into which the Emulator can connect. The quad connector scheme means that any target board can be connected to the emulator.

**Things to watch out for:**

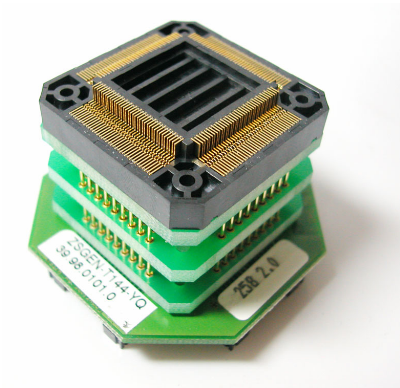
- The QuadConnect requires some board space which may rule it out in some designs. It is however very reliable and the unpopulated pads can be used for end-of-line testing in production.
- As there are effectively now two CPUs being driven, a discrete crystal clock circuit may not be sufficient. In such cases decreasing the load resistor value in the clock circuit may be needed or the DPROBEXC variable frequency clock generator can be used.  Note: if you contemplating not using a Hitex ICE, make sure that the unit you choose has a clock generator that can be easily varied, preferably by software.
- Do not add a pull-down resistor on P0.1 on your design as the DPROBEXC will do this for you.  Any additional resistor may cause problems!
- When deciding on the orientation of the CPU on the PCB, make sure that the body of the DPROBExc does not foul any components or other mechanical parts.

### 12.3.2 Yamaichi Socket

The Yamaichi IC149 sockets described earlier can also interface to the emulator. The emulator adaptor replaces the processor and retaining ring and requires longer screws.

Until the release of the PressOn adaptor, this was the most popular emulation connection method. Apart from requiring a CPU pad-length increase of 2mm, this method has relatively little impact on board design but will require the production of special socketed boards. The sockets are expensive and the emulator adaptor is more costly than the QuadConnect equivalent.

There are a number of different versions of the Yamaichi socket for a particular pin count and it is essential to get the right one.



**The Yamaichi Socket Adaptor**

| CPU | Package | Socket Part Number |
|---|---|---|
| XC164 | TQFP100 | IC149-100-025-B5 |
| XC161/7 | TQFP144 | IC149-144-045-B5 |
| | | IC149-144-145-B5 |

#### 12.3.2.1 Yamaichi Socket - Points To Watch Out For

- The socket requires pads that are 2mm longer than that required by a standard CPU and needs up to 4mm clearance around the CPU.
- There are sometimes two very similar IC149-series sockets of the same number of pins. Make sure you use the right one!

### 12.3.3 The Solder-In Stack

The solder-in “stack” or “replacement” adapter provides a reliable connection method but requires a rather tall (and expensive) block to be soldered into the CPU’s normal position. It is possible to fit a socket to the top of the stack so that the board can be run without the emulator but this then becomes physically very large. A major advantage over the socket method is that the stack will fit on standard-length pads.

### 12.3.4 Emulating Soldered-Down CPU’s

Emulation of soldered-down CPUs presents a particular problem as the conventional spring-contact “clip-over” connectors that were reliable with PLCC packages are almost totally useless on the MQFP and TQFP. The job of precisely locating up to 144 small spring-loaded terminals on a 0.5mm pitch is almost impossible. This has made the connecting of an emulator onto a production board with a surface mounted MQFP or TQFP processor a real challenge.

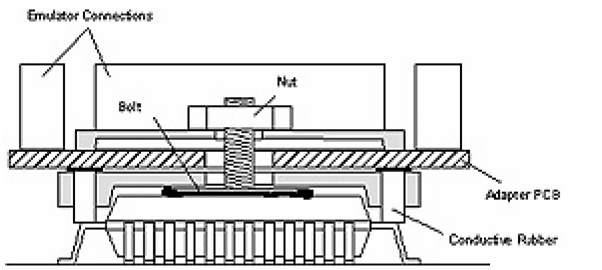
Hitex has developed a patented new technology based on narrow strips of a novel conductive elastomer that solves the connection problem. This special material is flexible and conducts only in one direction due to the alignment of its conductors. When pressed firmly against the shoulders of the CPU’s pins, it automatically aligns its conducting pathways to an interface board which are then directed to the emulator. All that is required is for the user to temporarily glue a threaded stud to the CPU, allowing the “PressOn” assembly to be clamped securely down by a nut.

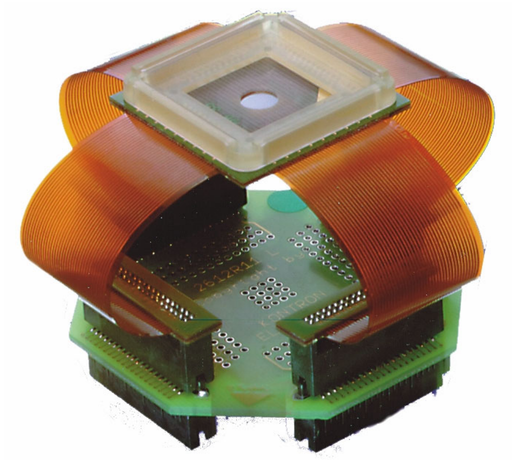
Where the Replacement and Yamaichi Adaptors require removing the target processor, the PressOn and Quad Connection schemes both leave the target processor in place. The emulator has to be able to tristate this processor. It does this by pulling P0.1 low during reset. This is no problem in external bus applications, but when

the processor is booting from the internal Flash, Port 0 is not used for configuration. The solution to this is that when the emulator senses that it is booting from internal flash, it momentarily resets the processor in external bus mode with P0.1 Low to tristate the target processor before resetting again back in internal bus mode. In order to do this the emulator must be able to pull the EA pin low so this pin should be pulled high through a resistor rather than directly to the 5v rail.

## 12.3.5 "PressOn" Emulation Adaptors

The "PressOn" was originally developed to meet the needs of the German car industry to allow soldered-down CPUs to be emulated without any changes being made to the target hardware design. Like the QuadConnect it relies on the soldered-in CPU tristating. The PressOn uses a conductive flexible elastomer material to make contact with the shoulders of the CPU's pins. This is similar to the flexible conductors used to connect flat screen LCD panels on portable PCs. The contact assembly is secured to the CPU package by using a special cyanoacrylate adhesive ("superglue") to attach a threaded stud. This glue is designed to be in tension so that the stud cannot be pulled off but be weak in shear, so that it can be easily twisted off with a scalpel. A solvent is then used to remove any glue residue. It is the (patented) technique of glueing a stud to the CPU package that gives the PressOn its reliability in real applications.



The PressOn Emulation Adaptor

PressOn frames are milled out from a sandwich of 6 different materials, including a flexible PCB layer. The CPU contact frame is machined from a rigid plastic material to ensure an accurate fit over the CPU package. The 0.5mm pin pitch used on the C16x family can be accommodated by current PressOn technology without problems.

The contact elements are largely hand-made and by using 6 gold conductors per millimetre the PressOn does not need precise alignment and guarantees a reliable connection to the target CPU. Despite the apparent fragility of the PressOn, at least 500 connection and disconnection cycles should be possible. If the contact elements do become damaged they can be replaced. To allow some relative movement between the CPU and and emulator the PressOn uses flexible sections.

All of this adds up to a rather expensive connection method but where there is insufficient space for a QuadConnect, it is the only alternative.

### 12.3.5.1 PressOn - Points To Watch Out For

There will need to be 4mm of clearance around the CPU pads to accommodate the PressOn contact frame and there must be no tall components within 25mm of the edge of the pads.

## 12.4 XC166 Family PCBs

### 12.4.1 Grounding Arrangements

Except in very low clock speed designs or possibly in educational projects, it is essential to use at least a gridded-ground earth plane. It is entirely possible to use a simple double-sided arrangement but it is usually the difficulty of routing up to 144 processor connections that dictates the use of a multi-layer board. At 40MHz though, the demands of low EMC emissions and reliability mean that at least a 4-layer or even 6-layer board will be required, with two power planes. It goes without saying that the clock source must be as physically close to the CPU as possible, as should be the memory devices. Unless a very large number of devices will be attached to the bus, no external bus drivers should be required. At 10-bits the A/D convertor inputs should be routed well away from the bus, preferably with each one interleaved with guard tracks. More detailed advice on handling the ADC can be found in chapter 9.

### 12.4.2 Electromagnetic Compatibility (EMC)

There are a number of very detailed application notes available covering general PCB layout issues, particularly with regard to EMC reduction. These can be found at www.infineon.com/xc166-family.

# 13 Getting New Boards Up-And-Running

Your new board arrives, fully assembled, with the microcontroller soldered down directly to the board. How do you get it running?

If you have designed your system using the guidelines set out earlier in this publication, then there is a good chance that the system will run straightaway. However experience has shown that it is better to take things one step at a time.

## 13.1 Useful Equipment

An oscilloscope is really an essential piece of kit when first testing new designs. However a proper in-circuit emulator or JTAG debugger is perhaps the greatest aid, as it allows you to effectively "sit" inside the CPU and look out across the bus - any bus errors are then obvious and a great deal of time can be saved. However, if you are lucky enough to have a DProbeXC in-circuit emulator (ICE) ready and waiting, do not plug it into the hardware straight away and switch on – XC Emulation Devices are delicate and expensive and a major board fault could destroy it. It is a very good idea to run through the basic checks listed below before jumping in with the emulator! Engineers equipped with a JTAG debugger like the TantinoXC can be a bit more cavalier, as these systems are rather more able to take short-circuits, Vcc on Vss pins etc..

For those without proper equipment, the bootstrap loader mechanism can be very useful for diagnosing hardware faults and the Infineon MINIMON bootstrap-loadable diagnostics tool is a free download. This is a very simple tool and can be a little tricky to use but it is infinitely better than nothing!. MINIMON will have to be used if there is no JTAG available, as might be the case on some XC164 designs as here the JTAG pins are shared with GPT1 functions.

### 13.1.1 (Almost) Free TantinoXC JTAG Debugger

If you are engaged in a commercial project, have the JTAG port available on your board but do not have access to a JTAG debugger then do not waste time using MINIMON! The new 99 Euro EasyKit for the XC164CM includes a 16k-code limited version of the TantinoXC JTAG debugger that in fact will work with any XC166 derivative. This makes commissioning new boards quite straightforward as it allows manual control of the CPU and external bus, includes a full C/C++ source level debugger plus it will program internal *and* external FLASH. The Hitop5 user interface is easy to use and can be upgraded later to remove the code limit, once the proper software development gets underway.

The EasyKit XC164CM is available to purchase at: http://www.ehitex.de


**XC164CM Easy Kit costs just 99 Euro and includes a powerful TantinoXC USB-JTAG debugger**

## 13.2 Before Applying Power…

It is definitely a good idea to check that the bus lines are not shorted to the power supplies *before* powering up the board. Making sure that the Vdd and Vss are connected to the right points on the CPU is also sensible. The analog reference and ground should also be checked as it seems to be quite common for these to be omitted or connected up incorrectly. These latter two points can spell instant damage to the emulation chip in a bondout-type emulator. However, should everything look OK, now would be a good time to plug the ICE into the hardware.

## 13.3 Testing The Board

### 13.3.1 External Start Applications

If your XC16 boots from external memory ((/EA = 0), make sure that the pull-down resistors on Port 0 are correctly installed, as any mistakes here will almost certainly prevent the CPU running properly. If you have done your pull-down resistor calculations properly then, with the XC held with the /RESIN pin low, there should be around 5v on any bus line that does not have a pull-down resistor and less than 1v on lines that have.

If your system has the EPROM socketed, then it is probably worth blowing a simple program into the EPROMs before fitting them to the board.

Where the FLASH EPROMs are soldered down and are empty, you will have to use the bootstrap loader or JTAG to initially get a test program into the board. The program need only wave a port pin up and down so that something can be seen on the 'scope. Make sure that the `while(1) { ; }` loop that contains the pin toggling code has a few NOPs in it, as if the loop is too small the CPU will simply jump within the instruction pipeline and the bus will appear to be dead. In the event of problems this inactivity could be misleading. If you are using the standard START_V2.A66 or CSTART.ASM C compiler start-up files, make sure you have altered the TCONCS0 register to program an appropriate number of waitstates as the Keil and Tasking default values may not be suitable. It is a good idea to enable the CLKOUT pin so that the real CPU frequency can be measured, in case the PLL is not working correctly. If at all possible, you should blow a test program into the EPROMs *before* they are soldered down as trying to program FLASH in-situ via the bootstrap loader on a brand new board, with a possibly unfamiliar CPU, may not be easy. This test program need only initialise a port as an output and then sit in a loop toggling it.

Powering up the board for the first time is always a slightly anxious moment. If your board is being powered off a bench power supply, turn the current limit down to say 250mA and wind it up slowly.....apart from the obvious of making sure that the current consumption is not excessive and that there is no smoke, some basic steps will have to be taken to confirm that the CPU can run. It is worth running a scope probe around the CPU pins to make sure that there is 5v and 2.5v on all of the appropriate pins and 0v on all the Vss and that there are no voltages above 5v on any pin. Now if you are lucky and you have a test program pre-installed in the EPROM, putting a 'scope onto P2.0 (or whatever your program in FLASH does) should reveal a square wave of around 2MHz. Should you be fortunate enough to get this, you are not quite home and dry because it is still possible for the program to run if you have the CPU running multiplexed in a non-multiplexed design. If you do not see anything then check the items in the next section. If your FLASH is empty you ought to check them as well but ultimately, you will have to use the bootstrap loader or a JTAG debugger to program it.

***Is the /RESIN pin high after powering-on?*** If your reset circuit is working correctly, it should be. If it is not, check the circuit! With the /RESIN pin high, the XTAL1/2 pins should show a clock signal of the frequency expected. The amplitude should be around 4v peak to peak if the RESIN pin is high. If it is low, the clock should have an amplitude of around 4.5v peak-to-peak. Changing the state of the /RESIN pin should change the clock amplitude by a selectable amount.

The ALE pin will be running, regardless of bus mode. Its frequency will give some idea of what the CPU is doing. If it is running with a high time of 50ns and a low time of 950ns, then the program is probably not being read correctly at all from the EPROM and the CPU is still running with its default 15 wait states. You may also see the CPU reset every 6.5ms (at 20MHz) with the RESETOUT going high and low as the on-chip watchdog trips

out. This will also cause the /CS0 to go high briefly. If your program successfully got through the initialisation code, the ALE will be running with a low time of around 150ns. It will thus have executed the EINIT instruction and so the /RESOUT pin will have gone high.

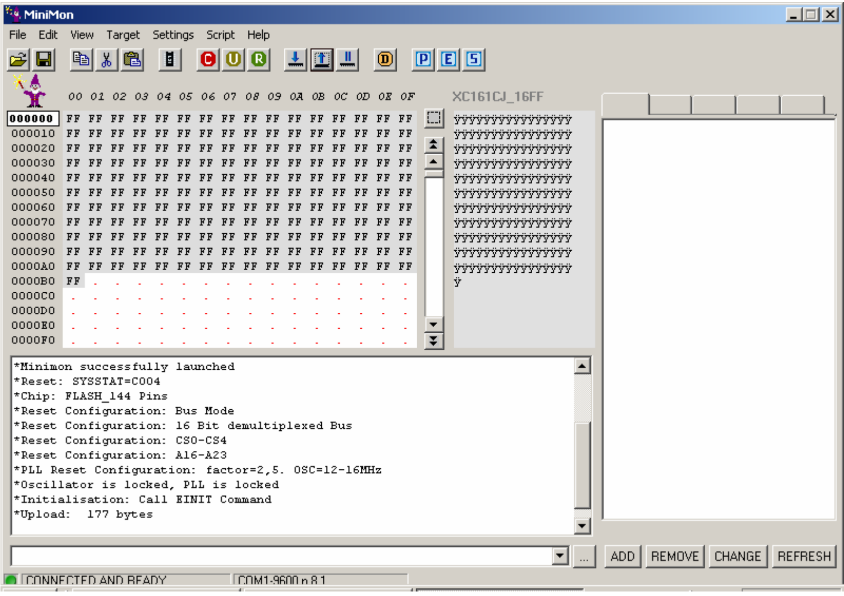
If nothing is happening on the ALE and the /RESIN is high, then check that there are no spurious pull-down resistors on Port 0.1 or 0.0 as these are the emulation modes and the chip will be in ONCE mode. Also check that the lower Port 0 lines are showing signs of activity.

Next, put the 'scope onto the /CS0 pin (P6.0) and check that it is high when the /RESIN pin is high and goes low when /RESIN is forced low. Make sure that the /CS0 makes it to the EPROM's /CE pin! Also check that the /RD pin is active after reset and that it is getting to the /OE on the EPROM.

If nothing unusual has been found, it is time to enlist the help of the CPU itself, either via its bootstrap mode or if you have access to the OCDS pins, using a JTAG debugger like the TantinoXC.

## 13.3.2 Using Serial Bootstrap Mode And MINIMON To Test New Boards

Infineon provide a simple but useful bootstrap-loaded diagnostics tool called "MINIMON", available from www.infineon.com/xc166-family). This allows you to look around inside the CPU and read back certain critical registers that can help sort out any problems. If your board has provision for using bootstrap mode, then make the link or whatever the mechanism is and power-cycle the board to put the XC into bootstrap mode. If you have not provided for this, put a 5k6 pull down resistor on the /RD pin (internal start) or Port 0, bit 4 (external start), and cause a reset. If you have no RS232 driver on serial port 0, you will have to add one, perhaps by putting a MAX232 on a piece of Veroboard and attaching the input lines to the S0TX and S0RX on the XC. On the connection to the PC COM port, pins 7 and 8 on the D-type connector will need to be connected together, as will 1, 4 and 6, so that the PC's UART will not hang up.



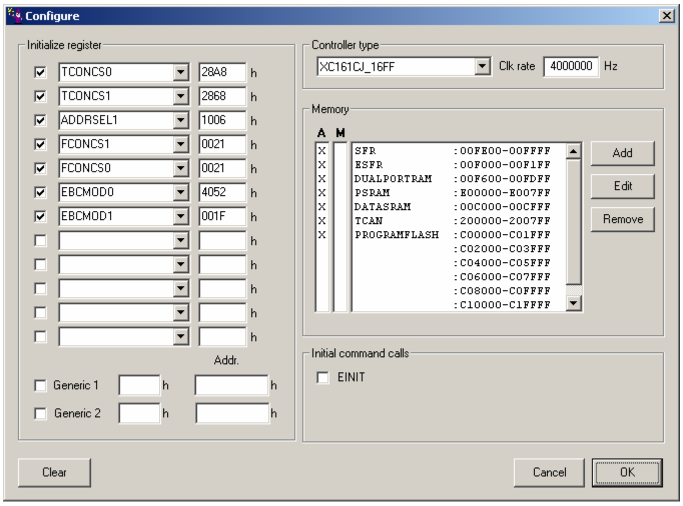

**CPU Hardware Configuration Read Back By MINIMON**

Using MINIMON utility, you should be able to get the CPU to report back the contents of the RSTCFG register. The contents of this register are latched from Port 0 at reset (see section 2.3.2) and from the returned value, you should be able to deduce the bus mode, number of address and chip select lines and whether the /WRH/WRL low mode is being used. All this information should be compared with the design specification for the board.

If you cannot get the CPU into bootstrap mode and the Port 0 bit 4 pull-down resistor is in place, it could be that the clock is unstable - simple crystal oscillators are prone to this, especially if you have not properly calculated the capacitor and load resistor values. It also might be that you have used too high a baud rate so make sure that you have not set it above 9600.

To check that external memory can be seen, use MINIMON to write some sensible values into the TCONCS0 and FCONCS0.   The target configuration will need setting up to contain at least these values for the registers controlling the external bus.

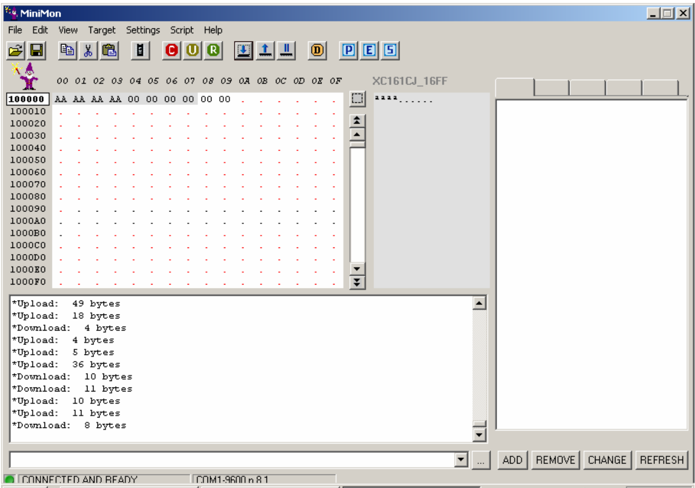
These values should allow you to see the external FLASH in the MINIMON upload window.  At this stage it is likely to contain just 0xFF's.  Any other value is likely to indicate a problem with the bus.



**Sensible Values For Testing The External Bus (16-bit demux)**

Now check the external RAM – here mapped to 0x100000.  Just writing 4 bytes to the RAM and reading it back successfully usually indicates that all is well.

If you are happy that the external bus is working then you could try programming something into the FLASH using MINIMON's external FLASH programming functions.  However it is much better to use MEMTOOL or a TantinoXC!



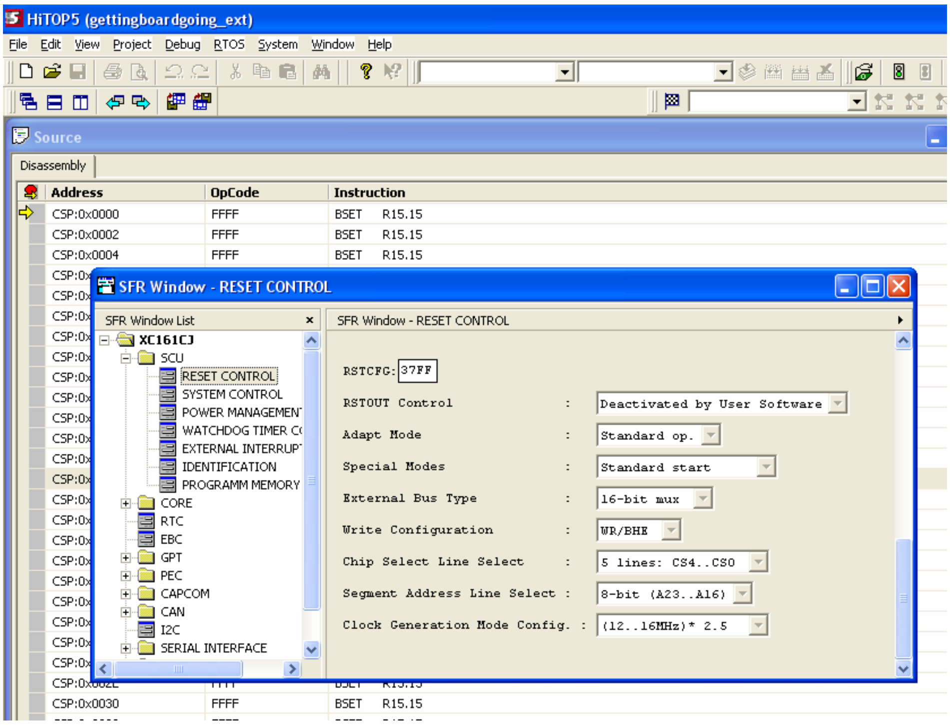
**Testing The External RAM**

### 13.3.3 Using JTAG For Testing New Boards

With the TantinoXC JTAG tool the procedure is much the same but a lot, lot quicker. The Hitop5 user interface is much easier to use, especially as regards setting up chip selects and FLASH programming. Before trying to use the TantinoXC on your new board, get it up-and-running on the XC164CM starter kit board. This is a "known good" target and you should be able to check out various windows and functions that will be needed to commission the new system.

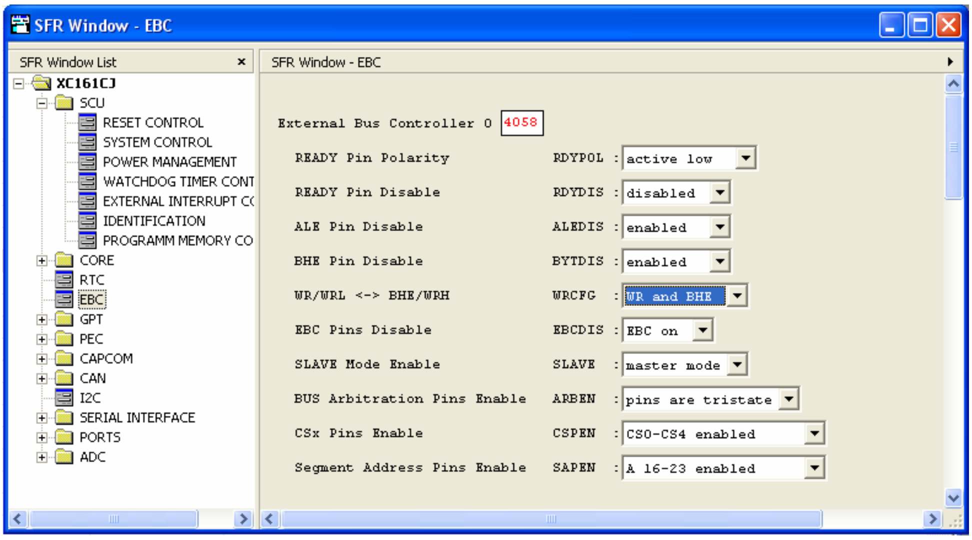
You will need to fit your chosen JTAG connector to the new board and check that the JTAG signals are routed correctly to the right CPU pins. If all is well, connect up the TantinoXC in the same way as with the starter kit board.

First test is to check the state of the RSTCFG register in the SFR window for SCU-RESET CONTROL. This will show up errors in the reading of the Port 0 configuration resistors.



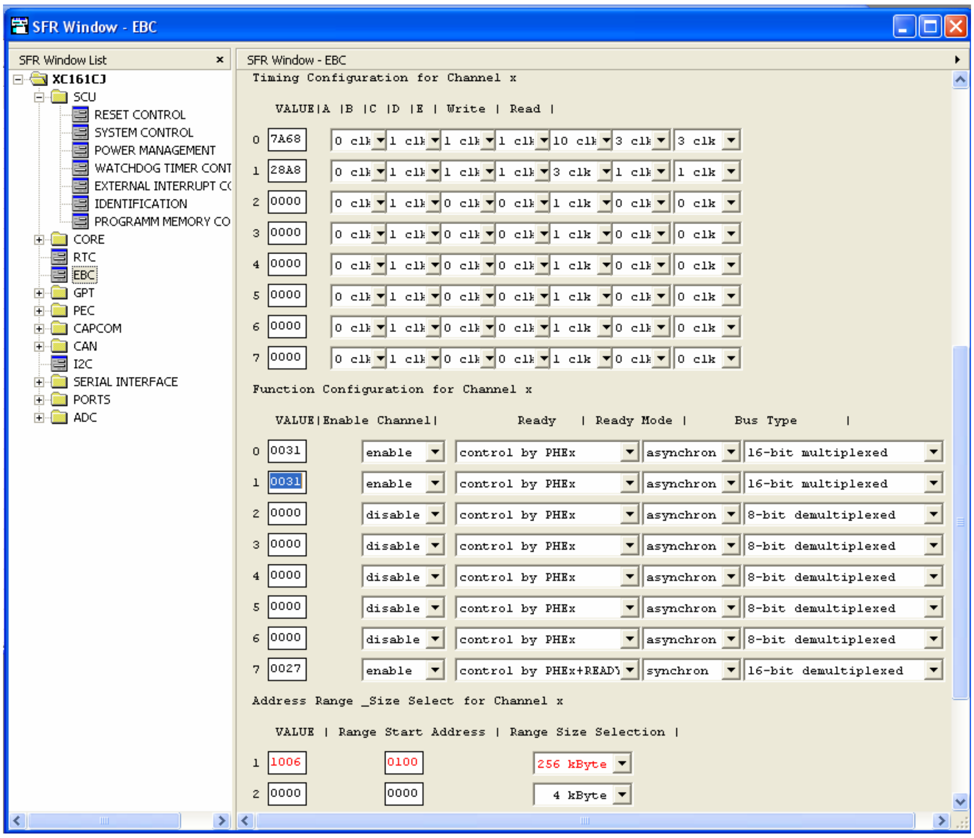

**RSTCFG Register Contents After RESET**

Next you will need to check that the external bus is operable. A good sign is that in the Hitop5 Disassembly window, you should see nothing but "BSET R15.5" instructions (as above). This indicates a blank FLASH with all data lines at least not stuck at zero and if the bus is multiplexed, that the latches are OK! Next the external RAM should be configured manually using the EBC SFR windows. This will allow the read/write operation of the bus to be checked.




**EDCMOD0 = 0x4058 For WR/BHE Mode Or 0x4858 For WRH/WRL Mode.**

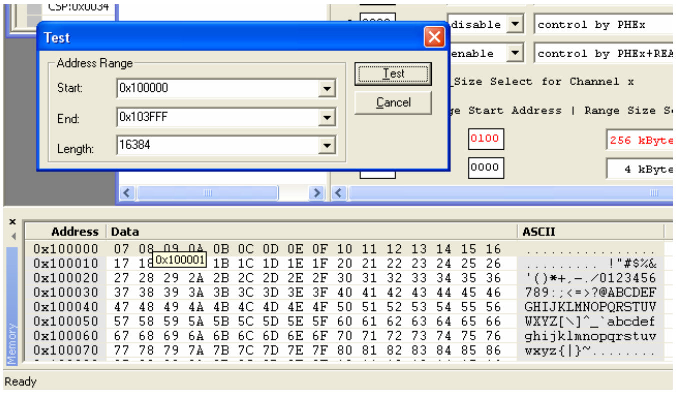
Assuming the RAM is on chip select 1, set up the TCONCS1, FCONCS1 and ADDRSEL1 registers by selecting the required settings from the dialog boxes. The values shown here will work for most RAMs, especially as the PLL will not be set up yet and so the CPU will be running at the external clock frequency divided by 2.




**Manually Configuring The External Bus (RAM mapped to 0x100000)**

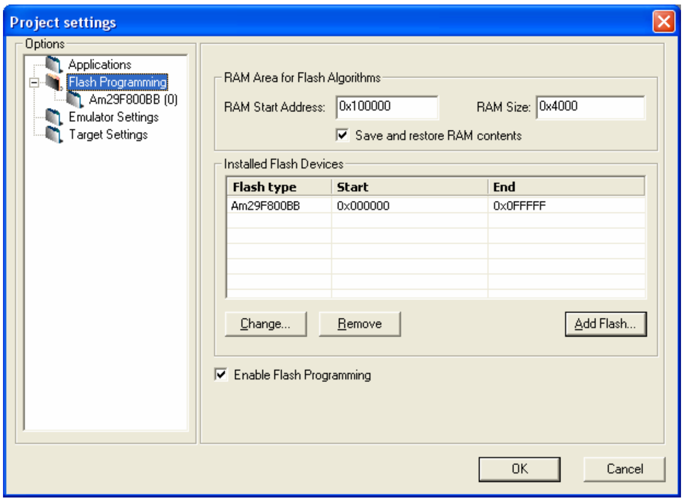
Now use the memory window to test the external RAM, now located at 0x100000 by the ADDRSEL1 setting of 0x1006.

This approach can be used to test external peripheral devices also controlled by chip selects.



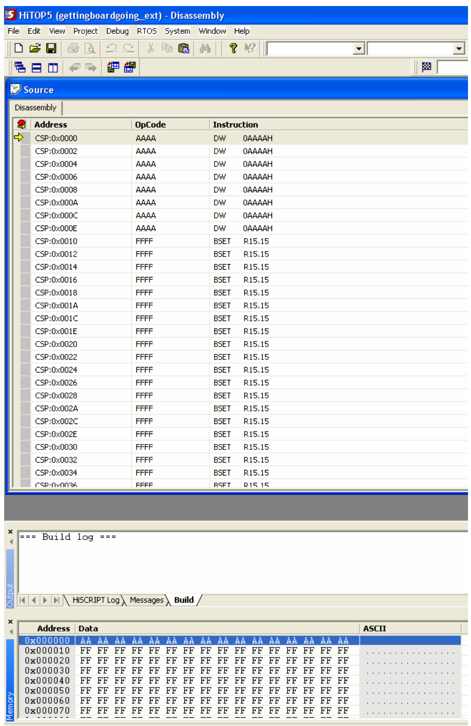

External RAM memory test

If no errors are reported then the next step is to test the external FLASH. First the TantinoXC FLASH programmer needs to be configured for the FLASH type on the board. Normally the RAM start address here would be the XC166's PSRAM at 0xE00000 but to test the external RAM, this was used to host the debugger's FLASH programming.




Configuring The TantinoXC FLASH Programmer

Then just write a test pattern (here "0xAA") into the first few FLASH locations via the memory window and the TantinoXC will blow the new data into the FLASH. Hopefully no FLASH errors will be reported. The most likely reason for failure is an incorrect setting of the TCONCS0 or FCONCS0 registers.



Writing A Test Pattern Into External FLASH

### 13.3.4 Common External Bus Problems

With a multiplexed bus, if you find that each location seems to contain a value identical to the lower byte of its address then it is likely that the demultiplexing is not working properly. If you can only write to the even bytes in a 16-bit bus system then suspect that there is a problem with the /WR & /BHE pins or that an 8-bit bus has been selected by accident. If the value of a location seems to change when you make several reads in succession then it is likely that you have the wrong bus timing for the memory device – the PhaseE setting is most likely to be too small.

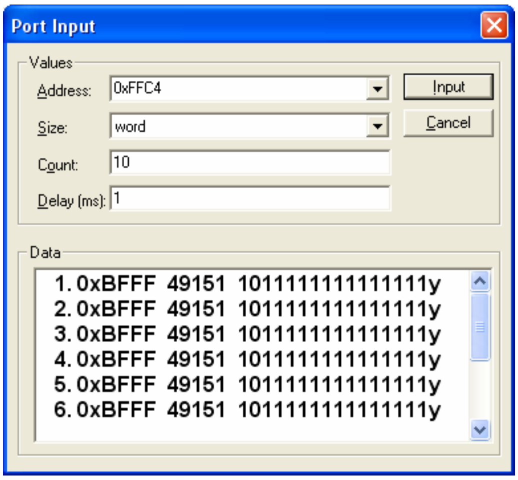
## 13.4 Internal Start Designs

With no external bus there is very little that can go wrong, other than the /EA pin being in the wrong state for single-chip operation. It can be very hard to see whether the CPU is in a workable condition with single-chip designs as there is no external bus to monitor. Therefore you will almost certainly have to use bootstrap mode or JTAG. If you are confident that the CPU is running, then you could put the device into serial bootstrap mode and use MEMTOOL to blow a test program into the on-chip FLASH and see if it works. However if you have a system with external memory as well, then it is a good idea to use the procedures in section 13.3 to check whether the external bus is working.

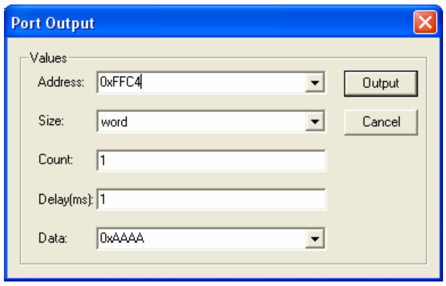
If your program will not run from the internal FLASH then use the TantinoXC to single step from the reset vector until you seem to lose control of the CPU. The last instruction executed is likely to be the one that contains a wrong setting! A commonly incorrectly-set item is the PLLCON (clock too high).

## 13.5 Testing The System

Having checked the operation of the external bus, the operation of the IO ports should be tested. Simple bit, byte or word writes to specific port pins will allow you to see whether the IO connections to the CPU are operational. Reading back port pin values likewise gives useful information.




**Multiple Reads Of Port 3**




**Writing A Test Pattern To Port 3**

The analog channels can be tested by using the SFR-ADC windows to make manually-controlled reads of the analog inputs.

If your FLASH is blank, now is the time to program it with a simple port pin-toggling program and make sure that the system will boot-up and run it successfully. If it does not, then suspect that the settings in the START_V2.A66 (Keil) or CSTART.ASM are wrong, especially the PLL settings or those for EBCMOD0, which will typically be 0x7800 or 0x7000 in an external start design, although you should check this for your particular system.

# 14 Conclusion

If you are about to embark on an XC166 family design, we hope that you will have found some useful hints and tips in this guide. Should you be evaluating the family for a new project, you should now have realised that behind the vast amount of information in the data books there is a really great processor.

Good Luck!

# 15 Acknowledgements

The authors would like to thank the following people for their help in producing this guide:

Karl Smith
Mike Copeland
Manfred Choutka
Joachim Klein
Steve Brown
Wendy Walker

*Plus all the hundreds of 166 family users in the United Kingdom...*

# 16 Feedback

We hope you find this guide useful. As we are constantly revising it, we would welcome your suggestions for revisions or new topics. If you have any clever XC166 tricks of your own, we would like to see them as well. Future editions will include such topics as EMC design, board layout, PC bus interfacing and others.

*Please email your suggestions to **INSIDEXC@HITEX.CO.UK***

# 17 Further Reading

If you enjoyed this XC hardware epic, you may like the software sequel “An Introduction To The C Language On The C166 Family”, available from Hitex for just $10.

There is also a complete “Teach Yourself XC Programming” self-study course available, including a powerful training board, including a local CAN network. This unique kit allows engineers to familiarise themselves with the XC CPU and its peripherals within their own workplaces. Please contact Hitex for more details.

# 18 Contact Addresses

*Published By:*

**Hitex Development Tools Ltd.**, University Of Warwick Science Park, Sir William Lyons Road, Coventry, CV4 7EZ, United Kingdom.

To request any of the example software mentioned in this guide, please email **INSIDEXC@HITEX.CO.UK** with your area of interest. If you have not seen what you want, it's probably worth contacting us anyway as we may well have produced something appropriate since this guide was published.

# 19 Appendix 1 - Infineon XC166 Family Part Numbers

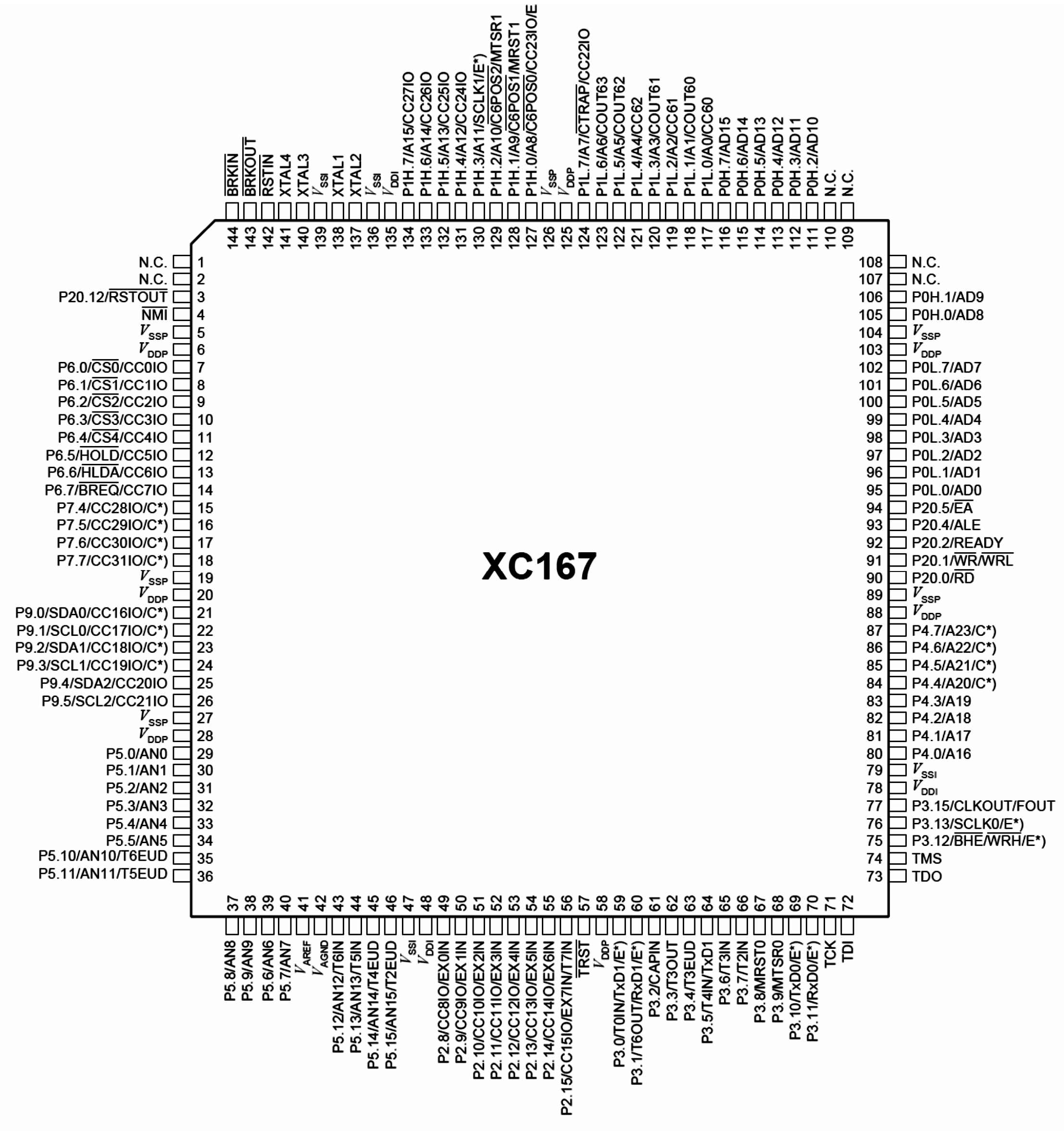
As with all electronic components care must be taken when ordering parts. In general **ALL** letters and digits of XC166 part numbers are significant and **MUST** be specified. If in doubt ask someone who knows!

**SAF-XC161CJ-16F20F**

| | |
|---|---|
| **Package Type:** | F = P-TQFP<br>M = P-MQFP<br>E = P-BGA |
| **CPU Speed:** | n*MHz |
| **Code Memory Type:** | R = ROM E = OTP<br>F = FLASH L = ROMless |
| **Code Memory Size:** | n * 8kbytes |
| **Key Features:** | C = CAN<br>J = J1850<br>U = USB<br>CS = 2x CAN nodes |
| **Core Type:** | XC16 = C166S V2<br>C16 = C166/7 |
| **Temperature Range:** | B = Commercial<br>F = Industrial<br>K = Automotive |

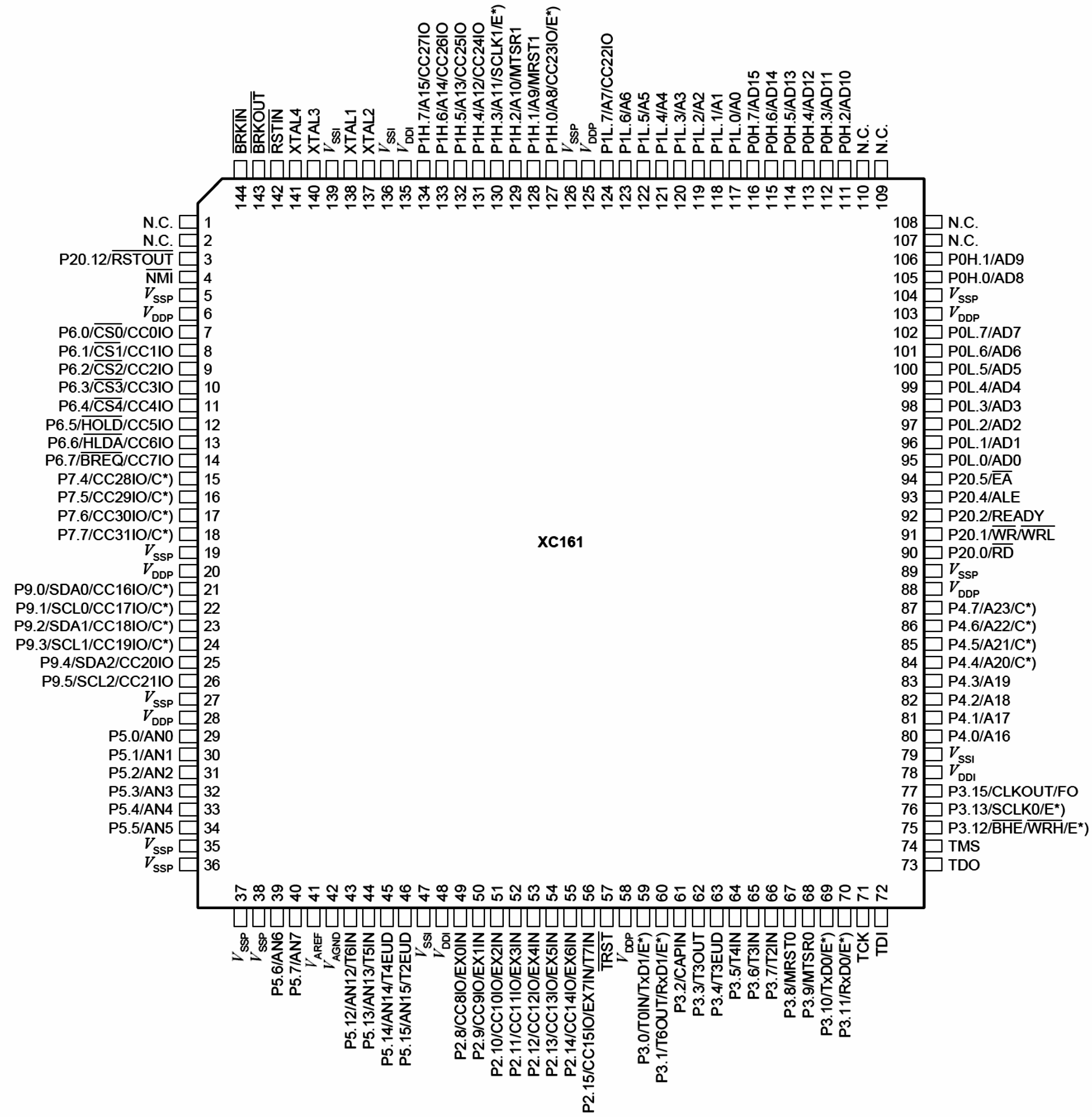
# 20 Appendix 2 – Pinout Of Common XC166 Derivatives

## 20.1 XC167CI Pinout



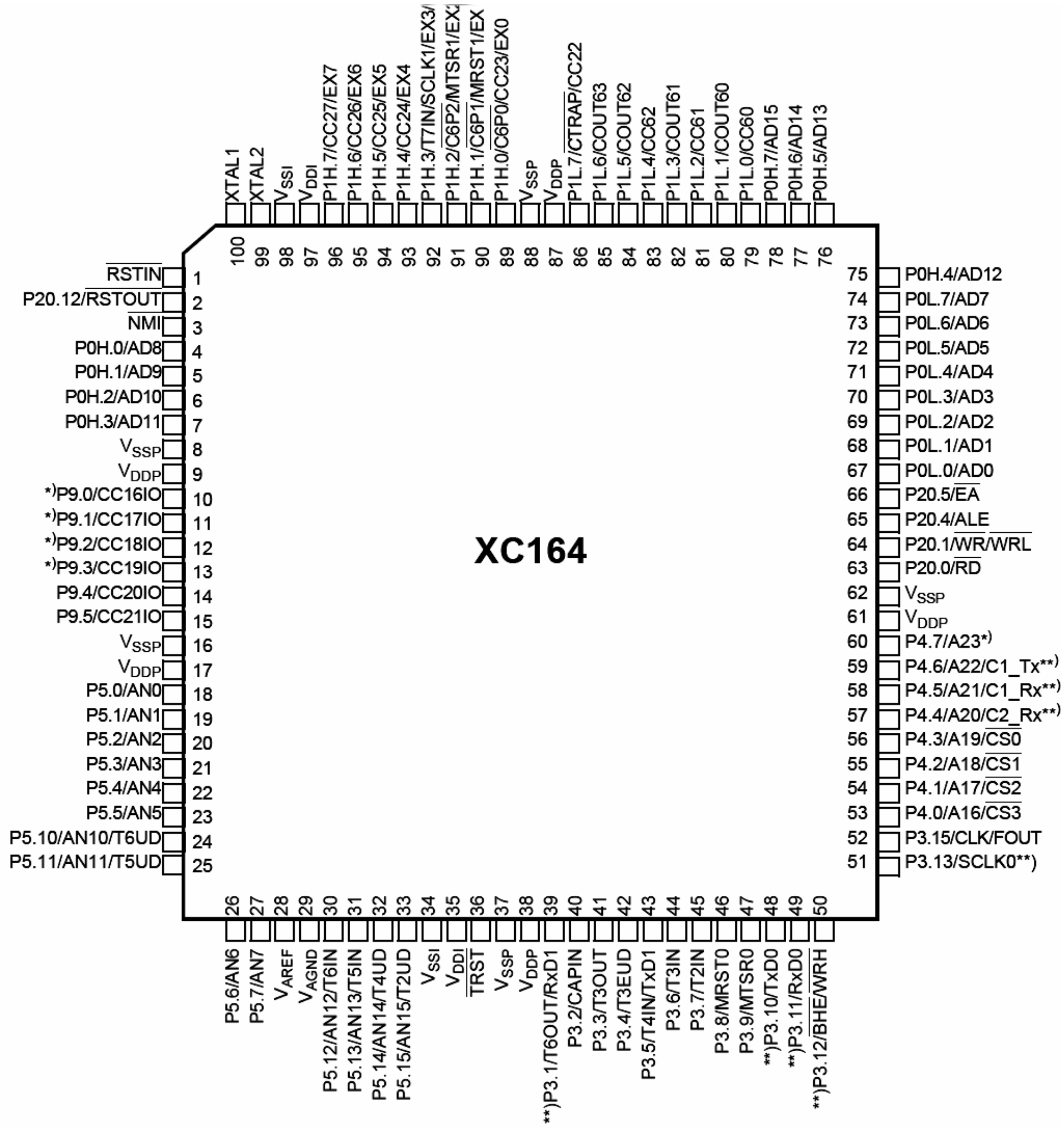

**Pinout Of P-TQFP-144 XC167CI**

## 20.2 XC161CJ Pinout



**Pinout Of P-TQFP-144 XC161Cx**

## 20.3 XC164Cx Pinout




**Pinout Of P-TQFP-100 XC164Cx**

www.infineon.com

Published by Infineon Technologies AG
Order No. B158-H8823-X-X-7600